Efficient Aggregation Algorithms for Compressed Data Warehouses

Jianzhong Li, *Member*, *IEEE Computer Society*, and
Jaideep Srivastava, *Senior Member*, *IEEE*

Abstract—Aggregation and cube are important operations for online analytical processing (OLAP). Many efficient algorithms to compute aggregation and cube for relational OLAP have been developed. Some work has been done on efficiently computing cube for multidimensional data warehouses that store data sets in multidimensional arrays rather than in tables. However, to our knowledge, there is nothing to date in the literature describing aggregation algorithms on compressed data warehouses for multidimensional OLAP. This paper presents a set of aggregation algorithms on compressed data warehouses for multidimensional OLAP. These algorithms operate directly on compressed data sets, which are compressed by the mapping-complete compression methods, without the need to first decompress them. The algorithms have different performance behaviors as a function of the data set parameters, sizes of outputs and main memory availability. The algorithms are described and the I/O and CPU cost functions are presented in this paper. A decision procedure to select the most efficient algorithm for a given aggregation request is also proposed. The analysis and experimental results show that the algorithms have better performance on sparse data than the previous aggregation algorithms.

Index Terms—Data warehouse, multidimensional array, OLAP, aggregation, aggregation on compressed data warehouses.

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 online analytical processing (OLAP). Aggregation and cube [1] are the most important operations of OLAP. The aggregation is used to "collapse" away some dimensions to obtain a more concise data set, namely, to classify items into groups and determine one value per group. The cube computes aggregations over all possible subsets of the specified dimensions. This paper aims at developing efficient aggregation algorithms for compressed data warehouses.

There are two kinds of data warehouses. One is for relational OLAP, called *ROLAP data warehouse* (*RDW* for short) [2], [3], [4]. The other one is for multidimensional OLAP, called *MOLAP data warehouse* (*MDW* for short) [5], [6], [7]. *RDW*s are built on top of relational database systems. *MDW*s are based on multidimensional database systems. A *MDW* is a set of multidimensional data sets. In a simple model, a *multidimensional data set* in a *MDW* consists of *dimensions* and *measures*, represented by

$$R(D_1, D_2, \ldots, D_n; M_1, M_2, \ldots, M_k),$$

where $D_i s$ are dimensions and $M_j s$ are measures. The data structures in which *RDW*s and *MDW*s store data sets are fundamentally different. *RDW*s 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, *MDW*s store their data sets as multidimensional arrays. *MDW*s only store the values of measures. The dimension values are treated as the indices of the multidimensional arrays. The position of the measure values within the multidimensional arrays can be calculated by the dimension values.

Methods of computing aggregation and cube for *RDW*s have been well-studied. A survey of the aggregation algorithms in relational database systems were presented in [11]. In [1], some rules of thumb were given for efficiently computing the cube for *RDW*s. In [12] and [13], algorithms were presented for deciding what *group-by*s to precompute for *RDW*s. In [14] and [15], a Cubetree storage organization for aggregation views of *RDW*s were proposed. In [16], fast algorithms for computing the cube operation on *RDW*s were given. These algorithms extend sort-based and hash-based methods with several optimizations. Aggregation precomputing 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 precomputed aggregations [19], and incrementally maintaining them [20].

While much work has been done on how to efficiently compute the aggregation and cube on *RDW*s, to the best of our knowledge, there is only one published paper on how to compute the cube on *MDW*s [10], and there is no published work on how to compute the aggregation on *MDW*s. *MDW*s present a different challenge in computing

- *J. Li is with the Department of Computer Science and Engineering, Harbin Institute of Technology, Harbin, 150001, Peoples Republic of China. E-mail: lijz@banner.hl.cninfo.net.*
- *J. Srivastava is with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455. E-mail srivasta@cs.umn.edu.*

Manuscript received 3 Aug. 1999; revised 19 July 2000 accepted 11 Dec. 2000; posted to Digital Library 7 Sept. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 110352.

the aggregation and cube. The main reason for this is the fundamental difference in physical organization of their data. *MDW*s normally have very large size and a high degree of *sparsity* that have made data compression a very important and successful tool in the management of *MDW*s. There are several reasons for the need of data compression. The first reason is that the multidimensional data sets in *MDW*s created by the cross product of the dimensions can be naturally sparse. For example, in an international trade data set 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 data set 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 of *MDW*s come from the properties of data values. Often the data values are skewed in some data sets, where there are a few large values and many small values. In some data sets, data values are large but close to each other. Also, sometimes certain values tend to appear repeatedly.

While there are many data compression techniques applicable for *MDW*s [8], [9], there is no work to date on how to compute aggregation directly on compressed *MDW*s, and there is only one published paper on how to compute the cube on compressed arrays [10]. Efficiently computing the aggregation and cube on compressed *MDW*s is a serious challenge since most large *MDW*s may be compressed for storage.

The goal of this paper is to develop efficient algorithms to compute aggregation on compressed *MDW*s. A set of aggregation algorithms on very large compressed *MDW*s is proposed in the paper. These algorithms operate directly on compressed data sets without the need to first decompress them and, therefore, are efficient for sparse data warehouses. The algorithms have different performance behavior as a function of data set 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 for a given aggregation request is also proposed. The analysis and experimental results show that the performance of the proposed algorithms is much better than the previous aggregation algorithms. The algorithms assume that *MDW*s are compressed by mapping-complete methods. Thus, they are only applicable to the mapping-complete compression methods that are common used in compressing *MDW*s. Now we are working on the aggregation algorithms that are applicable to other kinds of compression methods.

The rest of the paper is organized as follows: Some backgrounds are introduced in Section 2. In Section 3, we describe and analyze the proposed aggregation algorithms on compressed *MDW*s. In Section 4, we provide the decision procedure that selects the most appropriate algorithm for a given aggregation request. The experimental performance results are presented in Section 5, and the conclusions and future work are discussed in Section 6.

2 Backgrounds

The method to compress *MDW*s must be introduced before describing the aggregation algorithms for compressed *MDW*s. To compress a *MDW*, each data set in the *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 $R(D_1, D_2, \ldots, D_n; M_1, M_2, \ldots, M_m)$ be an n-dimensional data set with n dimensions, $D_1, D_2, \ldots, D_n$, and m measures, $M_1, M_2, \ldots, M_m$, where the cardinality of the ith dimension is d_i for $1 \leq i \leq n$. Using the *multidimensional array method* to organize R, each of the m measures of R is first stored in a separate array. Each dimension of R is used to form a dimension of all the 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 ith dimension of R is encoded into $0, 1, \ldots, d_i - 1$ for $1 \leq i \leq n$. The *array linerization function* for the multidimensional arrays of R is

$$LINEAR(x_1, x_2, \ldots, x_n) = x_1 d_2 d_3 \ldots d_n + x_2 d_3 \ldots d_n + \ldots + x_{n-1} d_n + x_n.$$

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

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

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

where,

$$y_n = Y \bmod d_n, y_i = [\ldots[Y/d_n]\ldots]/d_{i+1}] \bmod d_i$$

for $2 \leq i \leq n-1, y1 = [[\ldots[[Y/d_n]/d_{n-1}]\ldots]/d_3]/d_2]$. The array indices, $i_1, i_2, \ldots,$ and i_n, of the measure value in position P in a linearized array are determined by $R - LINEAR(P)$.

2.2 Data Compression

It is desirable to develop data compression techniques so that the data can be accessed in their compressed form and operations can be performed directly on the compressed data. Such techniques usually provide two mappings [8]. One is *forward mapping*.It computes the location in the compressed data set given a position in the original data set. The other one is *backward mapping*. It computes the position in the original data set given a location in the compressed data set. A compression method is called *mapping-complete* if

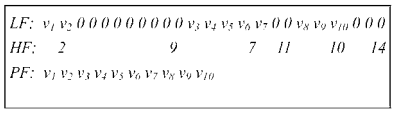

Fig. 1. An example data set compressed by header compression method.

it provides both *forward mapping* and *backward mapping*. Mapping-complete compression methods are often used methods for multidimensional databases and *MDW*s. Many compression techniques are mapping-complete, such as *header compression* [21], *BAP compression* [23], and *chunk-offset compression* [10].

The algorithms proposed in this paper are applicable to all the *MDW*s that are compressed by any mapping-complete compression method. To make the description of the algorithms more concrete, we assume that data sets in *MDW*s have been stored in linearized arrays, 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 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*. Fig. 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. The details of the header compression method can be found in [21].

3 Aggregation Algorithms

In the rest of the paper, without loss of generality, we assume that each data set has only one measure. Let $R(D_1, D_2, \ldots, D_n; M)$ be a multidimensional data set. A *dimension order* of R, denoted by $D_{i_1} D_{i_2} \ldots D_{i_n}$, 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 jth 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 $D_1 D_2 \ldots D_n$. The input of an aggregation algorithm includes a data set $R(D_1, D_2, \ldots, D_n; M)$, a *group-by dimension set* $\{A_1, A_2, \ldots, A_k\} \subseteq \{D_1, D_2, \ldots, D_n\}$ and an aggregation function F. The output of the algorithm is a data set $S(A_1, A_2, \ldots, A_k; F(M))$, where the values of $F(M)$ are computed from the measure values of R using 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 array of R.
- N_{oh}: the number of data items in the header of R.

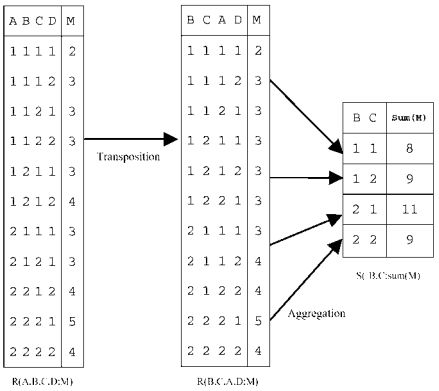

Fig. 2. Aggregation procedure of G-Aggregation.

- N_r: the number of data items in the compressed array of S.
- N_{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. This algorithm performs an aggregation in two phases. Phase one, called *transposition phase*, transposes the dimension order of the input multidimensional data set 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 4D data set stored in a linearized 4D array with dimension order $ABCD$. Assume that B, C is the group-by dimension set. The dimension order $BCAD$ and $BCDA$ are favorable dimension orders. In phase two, called *aggregation phase*, the algorithm computes the aggregation by one scan of the transposed R. Fig. 2 illustrates the algorithm. For expository purposes, we use the relational form in Fig. 2. In reality, the algorithm works directly on the compressed array of R.

The *transposition phase* assumes that W buffers are available. Data in the compressed array (physical file) is read into the buffers as much as possible. For each data item in a buffer, the following is done: 1) backward mapping is performed to obtain the logical position in the logical file, 2) the dimension values of the item are recovered using the reverse array linearization function, and 3) a new logical position of the item in the transposed space is computed using the array linearization function. The 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 in the increasing order of the tags. 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 are next merged in the increasing order of the tags using W buffers. A new header file is constructed for the transposed and 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 transposed, compressed, and linearized array of R, denoted by TA. The *aggregation phase* scans TA once, and

aggregates the measure values for each combined values of the group-by dimensions one by one.

G-Aggregation can be improved. To transpose the compressed array of R, G-Aggregation reads, writes and processes the run files (of the same size as that of the original compressed file) $\lceil \log_W \lceil \frac{N}{B} \rceil \rceil$ times in the transposition phase. To perform the final aggregation, another scan of the transposed array, the last run file, of R is needed. In each of the two phases, the original and transposed header files are accessed once. If the aggregation is performed as early as possible, the size of the run files will be reduced and the performance of the algorithm will be increase dramatically. To improve G-Aggregation, we perform aggregation and transposition 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 increasing order of the values of $A_1 A_2 \ldots A_k$. 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 array of R as possible, sorts them in the increasing order of the values of $A_1 A_2 \ldots A_k$, 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 array of R into the input buffer. For each data item v in the input buffer the following is done: 1) backward mapping is performed to obtain the logical position in the logical file, 2) the dimension values $\{x_1, x_2, \ldots, x_n\}$ of v are recovered using the reverse array linearization function, and 3) the values 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 in the increasing order of the tags. The sorted data items, each of which is in the form (v, *tag*) in the input buffer, are next locally aggregated with respect to their tags 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 aggregated and merged with respect to their tags again, 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 array of the aggregation result. Fig. 3 describes the main steps of the algorithm.

The improved G-Aggregation algorithm is described below. The input of the algorithm includes the compressed

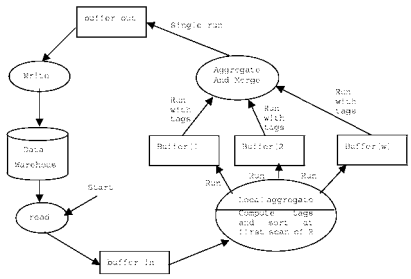

Fig. 3. Main steps of the improved G-Aggregation.

array *RPF*, stored in dimension order $D_1 D_2 \ldots D_n$, of $R(D_1, D_2, \ldots, D_n; M)$, the header file *RHF* of *RPF*, the group-by dimension set

$$\{A_1, A_2, \ldots, A_k\} \subseteq \{D_1, D_2, \ldots, D_n\},$$

and the aggregation function F. The output of the algorithm are the compressed array *SPF* of $S(A_1, A_2, \ldots, A_k; F(M))$ and the header file *SHF* of *SPF*.

Algorithm G-Aggregation
/* Phase one */
While *RPF* is not empty **Do**
***BuildNextRun*( );**
End While
/* Phase two */
While number of runs in ***RUN*** is greater than 1
Do /* ***RUN*** is the set of runs formed in phase one */
RUN'=empty set;
While more unprocessed runs in ***RUN*** **Do**
Use the *W* buffers to merge and aggregate the next *W* (or less) runs from ***RUN*** into a single run *r*;
Add *r* to ***RUN'***; /* the final aggregate and merge results are stored to *SPF* */
End While
RUN= RUN';
End While
Discard the logical position of each data, compute header counts and write to new header file *SHF*.
***BuildNextRun*( )**
AllBuffersFull=**False;**
While *RPF* is not empty and *AllBuffersFull* =**False Do**
Read next block from *RPF* into *buffer-in*;
For each value *v* in *buffer-in* DO
***ComputeAggregationDimensionValue*(*v*);**
End For
Sort *buffer-in* in the increasing order of tags;
Find minimun *j* from 1 to *W*, such that *buffer*[*j*] can take the contents of *buffer-in*
If such *j* is found **Then**
locally aggregate the data items with the same tag in *buffer-in* using *F*, copy the result to *buffer*[*j*];
Else *AllBuffersFull*=**True**;
End If
End While
Aggregate and merge the *W* buffers
buffer $[1], \ldots\ldots, buffer[W]$ in order of $A_1 \ldots A_k$

into a single run, write to *SPF*.

***ComputeAggregationDimensionValue*(*v*)**

Look up v's logical position using backward mapping function and header file *RHF*

compute the dimension values of v using reverse array linearization function and dimension order $D_1D_{2_n}$, store to D;

select the values $\{a_1, \ldots, a_k\}$ of the group dimensions $\{A_1, \ldots, A_k\}$ from D;

store $(a_1 \ldots a_k)$ with v to *buffer-in* as the tag of v.

3.1.2 Analysis of Cost

In the rest of the paper, the I/O cost of an algorithm is represented by the number of disk block accesses required by the algorithm. Because all the basic CPU operations can be executed in constant time, each of the basic CPU operations, i.e., addition, subtraction, multiplication, comparison, division, and data swap in memory, are called *computation*. The CPU cost is represented by the number of computations executed by the algorithm.

In the first phase, $\lceil N/B \rceil + (\lceil N_0/B \rceil - 1) + \lceil N_{oh}/B \rceil$ disk block accesses are needed to read the original compressed 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), where N_0 is the number of data items in all the runs generated in this phase. It is obvious that $N_0 \leq N$.

In the second phase, $\log_W S$ passes of aggregation and merge are needed, where S is the number of runs formed in phase one. Let N_i be the number of the output data items of the ith pass of aggregation and merge for $1 \leq i \leq \log_W S$, where $N_{\log_W S} = N_r$ is the number of data items in the compressed array of the final aggregation result. A buffering scheme is used so that in the odd (even) passes, 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 block accesses are required in this phase. In summary, the I/O cost of G-Aggregation is

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

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,

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

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

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

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

Now, 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 array of R, we need to perform a backward mapping and a reverse array linearization. Since the algorithm reads the compressed array of R from the beginning to the end, the header file only needs to be scanned one time from the beginning to the end to perform all the backward mappings. Thus, a backward mapping for each data item requires one computation. 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 for all values in the compressed array of R. $N - N_0$ 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 output, N_0 data items, of the first phase is generated by merging W buffers. In this way, generating an output data item requires at most W computations. Therefore, the total number of CPU operations for the first phase is

$$\begin{aligned}& 2N(n-1) + N + N - N_0 + \left\lceil \frac{N}{B} \right\rceil B \log_2 B + N_0 W = \\ & 2Nn - N_0 + \left\lceil \frac{N}{B} \right\rceil B \log_2 B + N_0 W.\end{aligned}$$

In the second phase, the algorithm performs $\log_W S$ iterations. The ith iteration involves the aggregating and merging of $\lceil \frac{S}{W^{i-1}} \rceil$ runs into $\lceil \frac{S}{W^i} \rceil$ and output N_i data items. In the ith iteration, $N_{i-1} - N_i$ aggregations are needed. The output, N_i data items, of the ith iteration is generated by merging W buffers. Each data item requires at most 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

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

In summary, the CPU cost of the algorithm G-Aggregation is

$$\text{CPUcost(G-Aggregation)} =$$
$$2Nn - N_0 + \left\lceil \frac{N}{B} \right\rceil B log_2 B + N_0 W + N_0 +$$
$$\sum_{i=1}^{\lceil \log_W S \rceil} N_i W = 2Nn + \left\lceil \frac{N}{B} \right\rceil B log_2 B + \sum_{i=0}^{\lceil \log_W S \rceil} N_i W.$$

Since $N_{i \le} \frac{1}{2^{i+1}}(N_r + N) + N_r$ on the average and $S \le \lceil \frac{N}{BW} \rceil$,

$$\sum_{i=0}^{\lceil \log_W S \rceil} N_i W \le W$$
$$\sum_{i=0}^{\lceil \log_W S \rceil} \left(\frac{N + N_r}{2^{i+1}} + N_r \right) \le (N + 2N_r + \lceil \log_W S \rceil N_r)$$
$$W \le \left(N + 2N_r + N_r \left\lceil \log_W \left\lceil \frac{N}{BW} \right\rceil \right\rceil \right) W$$

on the average. Thus, the average value of

$$\text{CPUcost(G-Aggregation)}$$

is

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

3.2 Algorithm M-Aggregation

This algorithm is superior to G-Aggregation in case that the aggregation result fits into memory. M-Aggregation computes aggregation by only one scan of the compressed array of the operand data set R. It reads blocks of the compressed array of R one by one. For each data item v in the compressed array of R, the following is done: 1) backward mapping is performed to obtain vs logical position; 2) the dimension values of v, $(x_1, x_2, \ldots, x_d)$, are recovered by the reverse array linearization function from the logical position of v, and the values $(a_1, a_2, \ldots, a_k)$ of the group-by dimensions are selected from $(x_1, x_2, \ldots, x_d)$; 3) if there is a w that is identified by $(a_1, a_2, \ldots, a_k)$ in the output buffer, aggregate v to w using aggregation function, otherwise insert v with $(a_1, a_2, \ldots, a_k)$ 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 with discarding of the tags. M-Aggregation is described as follows: The input of the algorithm includes the compressed array RPF, stored in dimension order $D_1 D_2 \ldots D_n$, of $R(D_1, D_2, \ldots, D_n; M)$, the header file RHF of RPF, the group-by dimension set

$$\{A_1, A_2, \ldots, A_k\} \subseteq \{D_1, D_2, \ldots, D_n\},$$

and the aggregation function F. The output of the algorithm are the compressed array SPF of $S(A_1, A_2, \ldots, A_k; F(M))$ and the header file SHF of SPF.

Algorithm M-Aggregation
For $i = 1$ TO $\lceil N/B \rceil$ **Do** /*size of *buffer-in* is B */
  Read the ith block of *RPF* into *buffer-in*;
  For each value v in *buffer-in* **Do**
    ComputeAggregationDimensionValue(v);
    /* this function is in G-Aggregation and result is $(a_1 \ldots a_k)$ */
    If there is no w in *buffer-out* whose tag is $(a_1 \ldots a_k)$ /* the **If-Then-Else** is implemented by hash method */
    Then insert v with tag $(a_1 \ldots a_k)$ in *buffer-out* in the order $A_1 \ldots A_k$;
    Else aggregation v to w in *buffer-out* using F;
  End For
End For
Write *buffer-out* to *SPF*, discard the tags, and build new header file *SHF*.

M-Aggregation requires one scan of the original compressed 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. Hence, the total I/O cost is

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

The CPU cost of M-Aggregation is the cost of, for each data item in the compressed array of R, performing a backward mapping, a reverse array linearization, a hashing computation, an aggregation or memory operation (move data to output buffer), and the cost for computing the result header counts. As discussed in Section 3.1.2, a backward mapping for each data item requires only one computation. All the backward mappings for all data items in the compressed array of R require N computations. All the reverse array linearizations for all data items require $2N(n-1)$ computations. The **If-Then-Else** sentence in the algorithm needs to be executed N times, each requires a hash computation. 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

$$\text{CPUcost(M-Aggregation)} =$$
$$N + 2N(n-1) + hN + N - N_r + N_r + N_r =$$
$$2Nn + Nh + N_r,$$

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

3.3 Algorithm Prefix-Aggregation

This algorithm takes advantage of the situation where the group-by dimension set contains a prefix of the dimension order $D_1 D_2 \ldots D_n$ of the operand data set $R(D_1, \ldots, D_n; M)$. It performs aggregation in main memory by one scan of the compressed array of R. It requires a main memory buffer large enough to hold each portion of the resulting compressed array for each "point" in the subspace composed by the prefix.

In the rest of the paper,

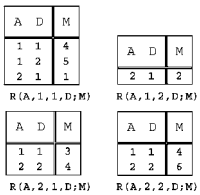

Fig. 4. Sample subsets of $R(A, B, C, D; M)$.

$$R(D_1, \ldots, D_k, a_{k+1}, \ldots, a_{k+p}, D_{k+p+1}, \ldots, D_n; M)$$

represents a subset of $R(D_1, \ldots, D_n; M)$ whose dimension values on $\{D_{k+1}, \ldots, D_{k+p}\}$ are $\{a_k, \ldots, a_{k+p}.\}$. For example, the subsets,

$$R(A, 1, 1, D; M), R(A, 1, 2, D; M), R(A, 2, 1, D; M)$$

and $R(A, 2, 2, D; M)$, of $R(A, B, C, D; M)$ in Fig. 5 are shown in Fig. 4.

We use an example to illustrate the idea of the algorithm. Assume that data set R has four dimensions A, B, C, and D, and is stored in a compressed array in dimension 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. Fig. 5 shows the idea of the algorithm. For each "point" (a, b) in the subspace, (A, B), of R, namely, (1,1), (1,2), (2,1), or (2,2), 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 this 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 described as follows: The input of the algorithm includes the compressed array RPF, stored in dimension order $D_1 D_2 \ldots D_n$, of $R(D_1, D_2, \ldots, D_n; M)$, the header file RHF of RPF, the group-by dimension set

$$\{A_1, A_2, \ldots, A_k\} \subseteq \{D_1, D_2, \ldots, D_n\},$$

and the aggregation function F. Please note that $A_1 A_2 \ldots A_p$ is a prefix of

$$D_1 D_2 \ldots D_n, p \leq k, A_{p+1} = D_{P+j_1}, A_{p+2} = D_{p+j_2}, \ldots, A_k = D_{p+j_k},$$

and $1 \leq j_1 < j_2 < \ldots . < j_k$. The output of the algorithm are the compressed array SPF of $S(A_1, A_2, \ldots, A_k; F(M))$ and the header file SHF of SPF.

Algorithm Prefix-Aggregation
For each point $(a_1, a_2, \ldots, a_p)$ in subspace $(A_1, A_2, \ldots, A_p)$ in increasing order **Do**
  For each block, *BLOCK*, of $R(a_1 \ldots, a_p, D_{p+1}, D_{p+2}, \ldots, D_n; M)$ **Do**
  Read *BLOCK* to *buffer-in*;
  For each value v in *buffer-in* **Do**
    ComputeAggregationDimensionValue(v); /*
    this function is in G-Aggregation and its output is $(a_1 \ldots a_k)$ */
    If there is no w in *buffer-out* whose tag is $(a_1 \ldots a_k)$
    Then append v with tag $(a_1 \ldots a_k)$ to *buffer-out* /*
    automatically in order of $A_1 \ldots A_k$ */
    Else aggregate v to w in *buffer-out* using F;
    End If
  End For
End For
Write the *buffer-out* to *SPF* with discarding of the "tags," and calculate related header counts and write to new header file *SHF*.
End For.

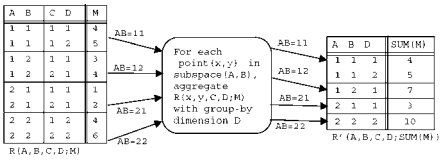

Fig. 5. Aggregation procedure of Prefix-Aggregation.

Prefix-Aggregation requires the reading of the original compressed array of R, writing of the resulting file, reading of the original header file and writing of the new header file. Hence, 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 the cost of performing, for each data item in the compressed array of R, a backward mapping, a reverse array linearization, a comparison, 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

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

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 $D_1 D_2 \ldots D_n$ of the operand data set

$$R(D_1, D_2, \ldots, D_n; M).$$

Fig. 6 shows the idea of the algorithm by an example of an aggregation with group-by dimension set C, D, which is a infix of the dimension order $ABCDE$, on data set $R(A, B, C, D, E; M)$. To perform the aggregation, R is first partitioned into four sorted runs,

$$R(1, 1, C, D, E; M), \quad R(1, 2, C, D, E; M), \quad R(2, 1, C, D, E; M), \text{ and } R(2, 2, C, D, E; M).$$

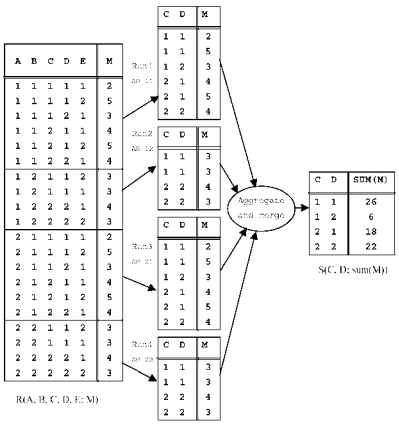

Fig. 6. Aggregation procedure of Infix-Aggregation.

Then, all the runs are projected onto (C, D) without removing repeated values. Finally, the projected runs are aggregated and merged to generate the aggregation result.

In general, let $\{D_{t+1}, D_{t+2}, \ldots, D_{t+k}\}$ be the group-by dimensions of an aggregation on $R(D_1, D_2, \ldots, D_n; M), n - k \geq t > 1, D$ be the size of the projection of R onto $(D_1, D_2, \ldots, D_t)$ and F be the aggregation function. It is obvious that $D_{t+1}D_{t+2}\ldots D_{t+k}$ is an infix of the dimension order $D_1D_2\ldots D_n$ of R. For each "point" $(a_1, a_2, \ldots, a_t)$ in $(D_1, D_2, \ldots, D_t)$, there is a sorted run,

$$R(a_1, \ldots, a_t, D_{t+1}, D_{t+2}, \ldots, D_n; M),$$

in the increasing order of the values of $D_{t+1}D_{t+2}\ldots D_{t+k}$. $R(D_1, D_2, \ldots, D_n; M)$ is the connection of D such runs. The Infix-Aggregation algorithm computes the aggregation by merging the D runs and performing aggregation at the same time. The algorithm assumes W buffers, each with size B, are available. If $W \geq D$,the algorithm requires only one scan of the compressed array of R. This algorithm is slower than Prefix-Aggregation when $W < D$ but is not as memory intensive as Prefix-Aggregation is. It requires $\log_W D$ passes to merge the D runs, where each pass merges W runs into one run. When 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,

$$R(a_1, a_2, \ldots, a_{t-1}, D_t, D_{t+1}\ldots., D_{t+k}; M)$$

represents the run for a "point" $(a_1, a_2, \ldots, a_t)$ in the subspace $(D_1, D_2, \ldots, D_t)$. The start position of

$$R(a_1, a_2, \ldots, a_t, D_{t+1}, D_{t+2}, \ldots., D_n; M)$$

in the compressed array of R can be computed by the following procedure.

run-position$(a_1, a_2, \ldots, a_t,$ *Header*$)$
Compute the logical position of $(a_1, a_2, \ldots, a_t, 0, 0, \ldots, 0)$, l, using array linearization function;
$Flag = 0$;
If $l \leq u_0$ **Then** $p = l$;
Else If $u_0 < l \leq c_0$**Then** $\{p = c_0 + 1; Flag = 1;\}$
Else Search for l on the sums of adjacent counts of *Header* to find i such that
$u_i + c_i < l \leq c_i + u_{i+1}$ or $c_{i-1} + u_i < l \leq u_i + c_i$
Endif;
If $Flag = 0$ and $u_i + c_i < l \leq c_i + u_{i+1}$**Then** {start position of $R(a_1, \ldots, a_t, D_{t+1}, \ldots., D_n; M)$ is $p = l - c_i$; exit; }
Else If $Flag = 0$ **Then** $p = u_i + c_i + 1$; **Endif**;
Endif;
compute the logical position of p, L, using backward mapping;
compute the dimension values $(b_1, \ldots, b_t, b_{t+1}, \ldots, b_n)$ of L using reverse array linearization function;
If $(b_1, \ldots, b_t) = (a_1, \ldots, a_t)$ **Then** start position of $R(a_1, \ldots, a_t, D_{t+1}, \ldots., D_n; M)$ is p;
Else $R(a_1, \ldots, a_t, D_{t+1}, \ldots., D_n; M)$ is empty;
End If;
End If.

Using the interpolation search [22], the I/O cost of the procedure is at most $2\log_2\log_2 N_h$ disk accesses and the CPU cost is at most $2\log_2\log_2 N_h + 4(n-1)$ where N_h is the number of data items in header file.

Infix-Aggregation starts by first computing the start positions of the D runs in the compressed array of R. Then, the algorithm performs 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 increasing order of the values of $D_{t+1}D_{t+2}\ldots D_{t+k}$. For the jth group $(1 \leq j \leq \lceil D/W \rceil)$, the algorithm reads as many blocks of each run in the jth group as possible, locally aggregates them, and fills one of the W buffers with local aggregation results. 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 jth new run. The process is repeated until all the data items in all runs of the jth group have been aggregated and merged into the jth new run. After the first iteration, the D runs are merged into $\lceil D/W \rceil$ sorted runs. In the following iteration, the ith iteration in general, the algorithm partitions the $\lceil \frac{D}{W^{i-1}} \rceil$ runs produced in the $(i-1)$th iteration into $\lceil \frac{D}{W^i} \rceil$ groups, each with W runs. For the jth group

$$\left(1 \leq j \leq \left\lceil \frac{D}{W^i} \right\rceil\right),$$

the blocks in the kth $(1 \leq k \leq W)$ run of the group pass through the kth buffer of the W buffers one by one. During the passing of the data items through the W buffers, the algorithm merges and aggregates the data in all the W buffers, and writes to the jth new run. In the last iteration, the aggregation result and its header file are generated. The Infix-Aggregation algorithm is described as follows: The input of the algorithm includes the compressed array RPF, stored in dimension order

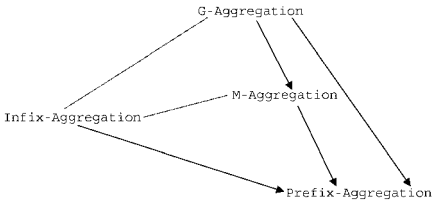

Fig. 7. Partial order of the algorithms.

$D_1D_2 \ldots D_n$, of $R(D_1, D_2, \ldots, D_n; M)$, the header file RHF of RPF, the group-by dimension set

$$\{D_{t+1}, D_{t+2}, \ldots, D_{t+k}\} \subseteq \{D_1, D_2, \ldots, D_n\},$$

the size D of the subspace $(D_1, D_2, \ldots, D_t)$, and the aggregation function F. The output of the algorithm are the compressed array SPF of $S(A_1, A_2, \ldots, A_k; F(M))$ and the header file SHF of SPF.

Algorithm Infix-Aggregation
For each $(a_1, \ldots, a_t)$ in $(D_1, \ldots, D_t)$ in increasing order
  Do /* compute the start positions of D runs */
  run-position$[a_1, \ldots, a_t]$:= *run-position*$(a_1, \ldots, a_t, RHF)$;
/* The first iteration */
For $i = 1$ **To** $\lceil D/W \rceil$ **Do**
  ***Build-i*th-*Run*(*i*)**; /* merge and aggregate runs $((i-1)W+1), ((i-1)W+2), \ldots, \text{and}(iW)$ into the ith new run */
End For
/* The rest $\lceil \log_W D \rceil - 1$ **iterations */**
While number of runs in ***RUN***is greater than 1 **Do**
  /* *RUN* is the set of $\lceil D/W \rceil$ runs formed in the first iteration */
  RUN'*=*empty set;
  While more unprocessed runs in ***RUN*** **Do**
    Use the W buffers to merge and aggregate the next W (or less) runs from ***RUN*** into a single r;
    Add r to ***RUN'***; /* the aggregate and merge results is stored to *SPF* */
  End While
  RUN*=*RUN';
End While
discard the tags of all data items, compute header counts and write to new header file *SHF*.
***Build-i*th-*Run*(*i*)**
While one of the $((i-1)W+1)\text{th}, ((i-1)W+2)\text{th}, \ldots,$ and $(iW)\text{th}$ runs is not empty *Do*
  For $j = 1$ **To** W **Do**
    While the $((i-1)W+j)\text{th}$ run is not empty and *buffer*[j] is not full **Do**
      Read next block from the $((i-1)W+j)\text{th}$ run into *buffer-in*;
      For each value v in *buffer-in* **Do**
        ComputeAggregationDimensionValue (v); /* the same as in G-Aggregation and result is $(a_1 \ldots a_k)$ */
      End For
      locally aggregate the data items with the same tag $(a_1 \ldots a_2)$ in *buffer-in* using F, copy the result to *buffer*[j];
    End While
  End For
  Aggregate and merge the W buffers *buffer*[1],. . . . . ,*buffer*[W] in the order $A_1 \ldots A_k$ into the ith new run, write to *SPF*;
End While.

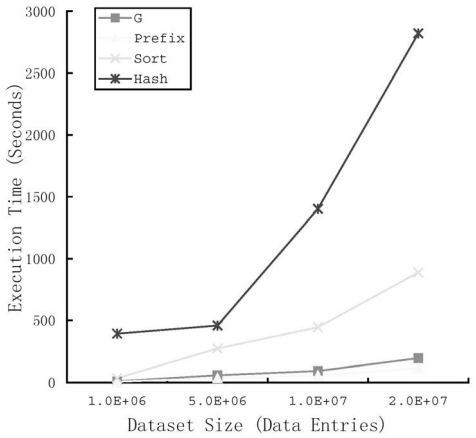

Fig. 8. Comparisions of Hash, Sort, G, and Prefix with varying data set size.

3.4.2 Analysis of Cost

Similar to the cost analysis of G-Aggregation, the average I/O cost of Infix-Aggregation is

$$\text{AIOcost(Infix-Aggregation)} = \text{O}\left(2D\log_2\log_2 N_{oh} + \left\lceil\frac{N}{B}\right\rceil + \left\lceil\frac{N_r}{B}\right\rceil + \left\lceil\frac{N_{oh}}{B}\right\rceil + \left\lceil\frac{N_{rh}}{B}\right\rceil + 2\left(\frac{N}{B} + \lceil\log_W D\rceil\frac{N_r}{B}\right)\right),$$

and the average CPU cost of Infix-Aggregation is

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

4 Comparison of Algorithms and Selection Procedure

Let X and Y be two algorithms. We use $X \geq_{\text{cost}} Y$ to represent the fact that the I/O and CPU cost of X is greater than or equal to that of Y. Similarly $X \geq_{\text{CPU}} Y$ denotes that the CPU costs of X are larger than that of Y. From the functions of I/O and CPU costs of the algorithms proposed in the paper, we have the following observations: Since all the justifications are very simple, we ignore all of them.

Observation 1.
G-Aggregation$\geq_{\text{cost}}$ Prefix-Aggregation
G-Aggregation$\geq_{\text{cost}}$ M-Aggregation,
M-Aggregation, $\geq_{\text{cost}}$ Prefix-Aggregation, and
Infix-Aggregation $\geq_{\text{cost}}$ Prefix-Aggregation.

Observation 2.

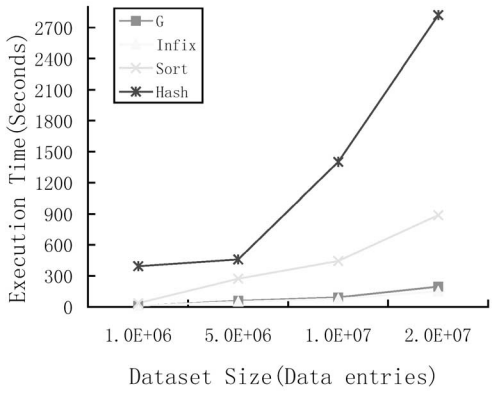

Fig. 9. Comparisons of Hash, Sort, G, and Infix with varying data set size.

If Infix-Aggregation$\geq_{\text{CPU}}$ M-Aggregation, then Infix-Aggregation $\geq_{\text{cost}}$ 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. Furthermore, Prefix-Aggregation places special requirements on the group-by dimensions.

Fig. 7 presents the order determined by Observation 1. Each directed edge expresses a relation "$\geq_{\text{cost}}$." A dashed edge between two algorithms represents no cost domination relation can be determined between the two.

Below, a general decision procedure is given that is based on the partial order graph in Fig. 7. In the procedure, α 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 represents "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

$$\mathbf{A} = \text{Infix-Aggregation} \geq_{\text{cost}} \text{G-Aggregation}$$
$$\mathbf{B} = \text{Condition of Observation 2, and}$$
$$\mathbf{C} = \text{Infix-Aggregation} \geq_{\text{cost}} \text{M-Aggregation}.$$

Algorithm SELECT
If $\gamma \wedge \eta$ **then** select Prefix-Aggregation;
else
  If $\beta \wedge$ ((B or C) or (not α)) **then** select M-Aggregation;
  else
  If $\alpha \wedge$ (not A) **then** select Infix-Aggregation;
  else select G-Aggregation.
  Endif
Endif
EndIf

The SELECT procedure is dependent on the I/O and CPU costs of the aggregation algorithms. All the costs required by the SELECT procedure can be determined using the cost functions in Section 3.

5 Experimental Rerults

To examine the performance of the aggregation algorithms, in practice, all the four aggregation algorithms in Section 3 have been implemented using C in the Windows NT 4.0 environment on a Gateway 2000 E3200 PC computer with Pentium II 350 CPU, 256MB memory and IBM-DATA-371010 disk system. The logical size of a disk block is 4k bytes.

To compare with the aggregation algorithms in relational database systems, we also implemented the sort and hash-based aggregation algorithms [11]. The

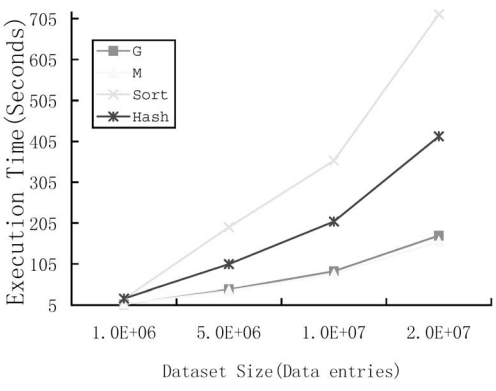

Fig. 10. Comparisons of Hash, Sort, G, and Infix with varying data set size.

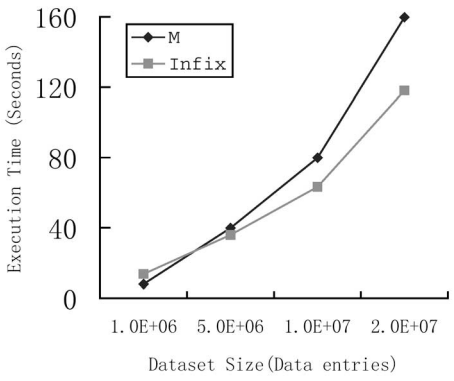

Fig. 11. Comparisons of Hash, Sort, G, and Infix with varying data set size.

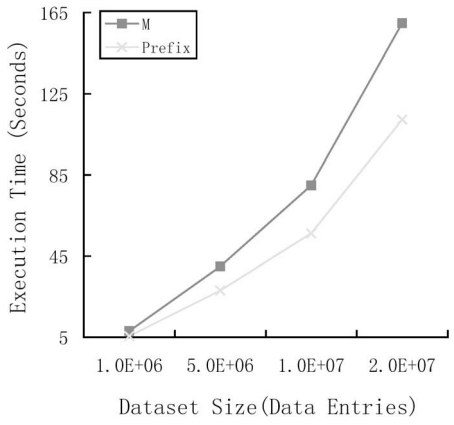

Fig. 12. Comparisons of Hash, Sort, G, and Infix with varying data set size.

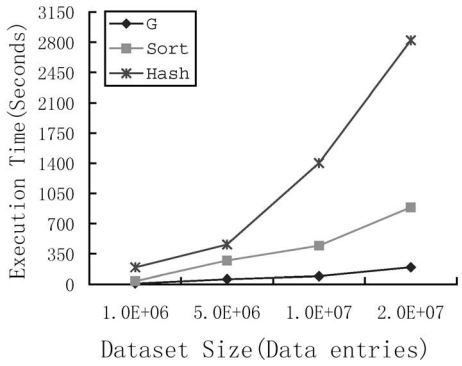

Fig. 13. Comparisons of Hash, Sort, G, and Infix with varying data set size.

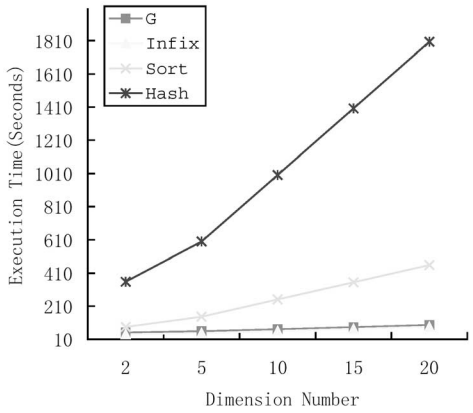

Fig. 15. Comparisions of Hash, Sort, G, and Infix with varying dimensions numbers.

experimental results show that our algorithms have much better performance than the previous algorithms.

There are four factors that affect the performance of aggregation algorithms. The first one is *size of data set*. The second one is *compression ratio*. It 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 size of the extra storage space is the header size. The third one is the size of *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, data sets were randomly generated, and stored using the compressed method described in Section 2 for our algorithms and relational tables for the traditional aggregation algorithms. In each experiment, we randomly generated 10 aggregation operations, let each algorithm perform all the 10 operations, and then 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 Size of Data Set

In these experiments, the benchmark data set 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 data set is 50 percent of the data set size. The aggregation result size of each data set is 20 percent of the data set 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. Fig. 8 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 data set 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 data set 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

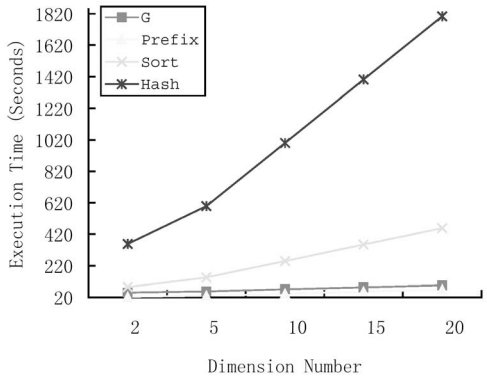

Fig. 14. Comparisons of Hash, Sort, G, and Prefix with varying dimension numbers.

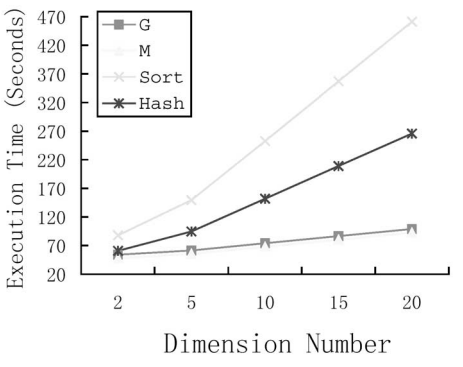

Fig. 16. Comparisons of Hash, Sort, G, and M with varying dimensions numbers.

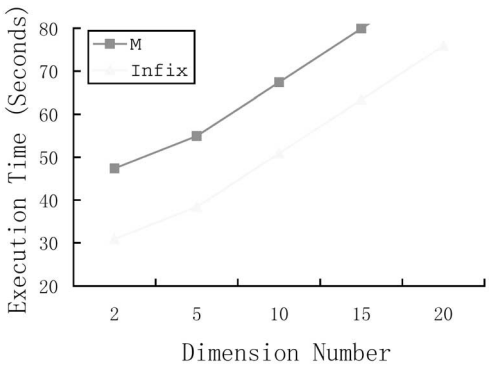

Fig. 17. Comparisons of M and Infix with varying dimensions numbers.

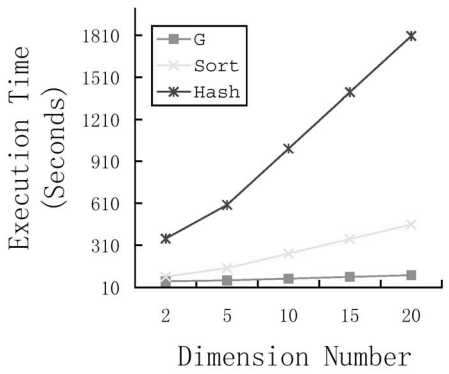

Fig. 19. Comparisons of Sort, Hash, and G with varying dimension numbers.

size can hold the whole aggregation result when the data set 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 acceptable time using Hash algorithm, the available memory size is set to 20 percent of the aggregation result size for Hash. Fig. 9 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 percent of the maximum data set size 20,000,000, so that all the aggregation results fit in memory. Fig. 10 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 execution times of M when the data entry number is smaller than 5,000,000. It is because that when the data set 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. Fig. 11 presents the comparisons of M and Infix, and Fig. 12 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 any restriction. The available memory size was fixed at 640K. The experiment results in Fig. 13 show Hash>Sort>G.

5.2 Performance Related to Data Compression Ratio

We first conducted experiments to study the performance of the algorithms while the data set 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 data sets was varied from two 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 M. In the first two sets of experiments, the available memory size is fixed at 12.5M bytes.

In the first set of experiments, the aggregation operations performed satisfy the requirements of Prefix. Fig. 14 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 data set does not affect the size of the compressed array storing the data set so that the I/O costs of G and Prefix vary slightly when the

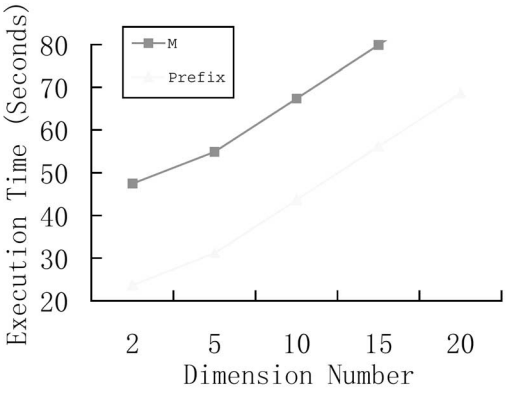

Fig. 18. Comparisons of M and Prefix with varying dimensions numbers.

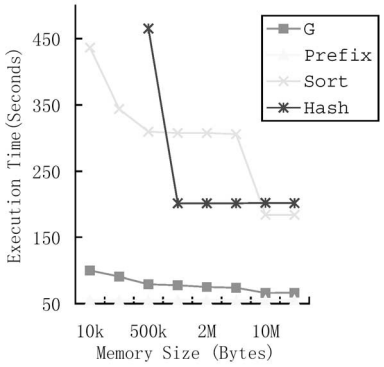

Fig. 20. Comparisons of Sort, Hash, Prefix, and G with varying memory sizes.

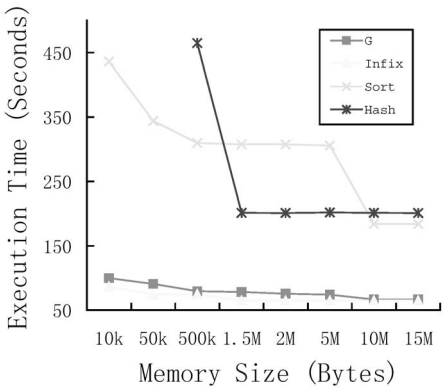

Fig. 21. Comparisons of Sort, Hash, Infix, and G with varying memory sizes.

number of the dimensions increases. On the opposite, when the dimension number of a data set increases, the size of the relational table storing the data set increases. Thus, the I/O costs of Hash and Sort increase very fast with the increasing of the dimension number. Fig. 14 also show Hash>Sort>G>Prefix.

In the second set of experiments, the aggregation operations performed satisfy the requirements of Infix. Fig. 15 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. Fig. 16 presents the experiment results. The figure shows that the execution times of Sort and Hash increase much faster than M and G, the performance of M is only a little better than G in case of aggregation results fitting in memory.

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. Fig. 17 presents the comparison of M and Infix, and Fig. 18 presents the comparison of M and Prefix.

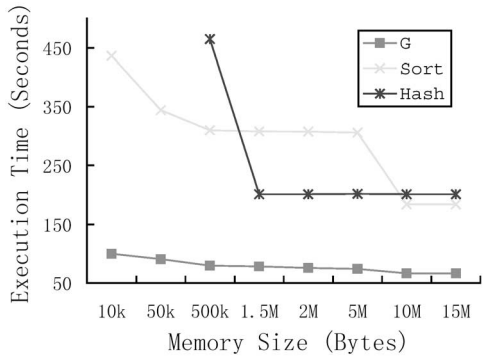

Fig. 22. Comparisons of Sort, Hash, and G with varying memory sizes.

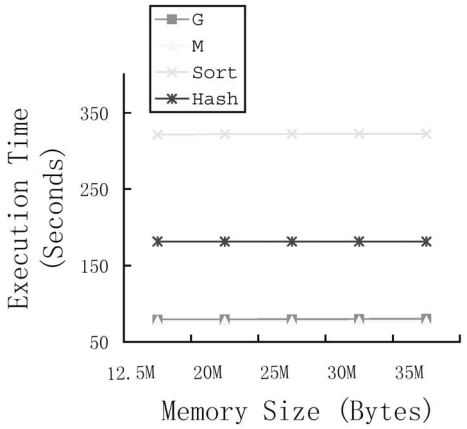

Fig. 23. Comparisons of Sort, Hash, M, and G with varying memory sizes.

In the fifth set of experiments, memory size is fixed at 12.5M bytes. Fig. 19 presents the experiment results. The figure indicates that the execution times of G increase much slower 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 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 [24].

5.3 Performance Related to Memory Size

We conducted five sets of experiments to study the performance of the algorithms while the memory size varies. In all the experiments, each operand data set consists of 15 dimensions, one measure, and has 10,000,000 data entries, each with 64 bytes.

In the first three sets of experiments, the aggregation result size is fixed at 2,000,000 data entries, and the available memory size varies from 10k bytes to 15M bytes. Figs. 20, 21, and 22 illustrate the results of the three sets of

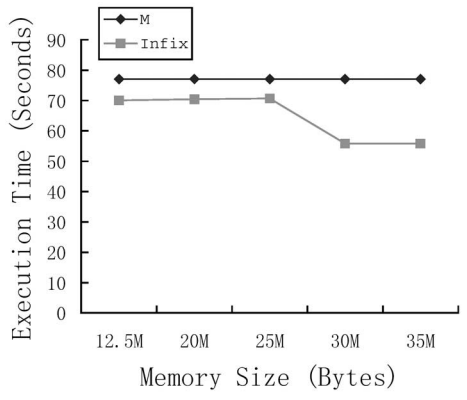

Fig. 24. Comparisons of M and Infix with varying memory sizes.

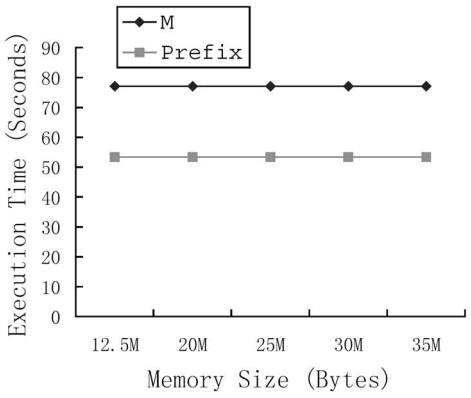

Fig. 25. Comparisons of M and Prefix with varying memory sizes.

experiments. The figures indicate that the execution time reduces while the available memory size increases, after available memory size reaches some point (here is 1.5M bytes) the execution times of all the algorithms reduce slowly, and Hash is very sensitive to available memory size.

In the fourth and fifth sets of experiments, the aggregation result size is fixed at 200,000 data entries, and the memory size varies from 12.5M to 35M bytes. Memory of 12.5M bytes can hold the aggregation result that meets the requirement of M. Figs. 23, 24, and 25 present the results of the fourth set of experiments. Since the aggregation results can be held in memory, all algorithms become main memory algorithms. Thus, the execution times of all the algorithms vary slightly while the memory size increases. Fig. 24 shows that the execution time of Infix reduces significantly when the memory size is greater than 25M bytes. The reason is that after the memory size is greater than 25M the buffer number is greater than the number of runs that need to be merged so that Infix can merge the runs in one scan of the operand data set.

5.4 Performance Related to Dimension Size

The dimension size, namely, the number of elements in each dimension, has a significant effect on the performance of Infix. When the dimension size is increased, the number of the runs that need to be merged by Infix may increase so that the I/O and CPU cost may increase as well. However, the dimension size has a little effect on other aggregation algorithms. Since Infix and Prefix are used for different

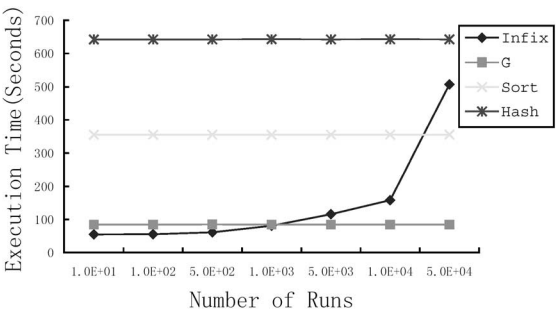

Fig. 26. Comparsion of G, Infix, Sort and Hash with varying run numbers.

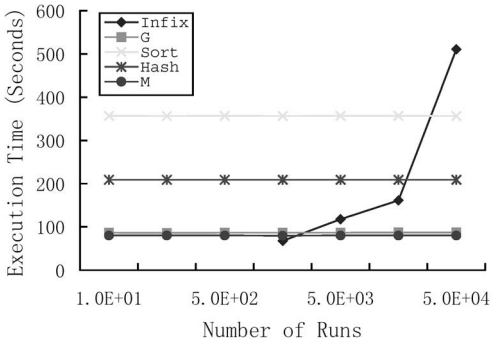

Fig. 27. Comparison of G, M, Infix, Sort, and Hash with varying run numbers.

aggregation operations, we cannot compare them. We conducted two sets of experiments to study the performance of the aggregation algorithms while the dimension size varies. In these experiments, all operand datsets consist of 15 dimensions and one measure. Its first two dimensions have fixed sizes 2 and 5, the size of the third dimension varies from 1 to 5,000, and every aggregation dimension set, which is an infix of the dimension order of the operand data sets, of each aggregation operation begins from the fourth dimension. Thus, the number of the runs that need to be merged by Infix varies from 10 to 50,000. In the experiments, the data set size is fixed at 10,000,000 data entries, each with 64 bytes, the aggregation result size is fixed at 2,000,000 data entries, and the run number varies from 10 to 50,000.

In the first set of experiments, available memory size is 32M bytes. Fig. 26 illustrates the execution times of G, Infix, Sort, and Hash while the run number varies. In the second set of experiments, the aggregation result can be held in available memory, namely, available memory size is 125M bytes. Fig. 27 shows the execution times of G, M, Infix, Sort and Hash while the run number varies. The figures show that the performance of Infix will be worse than other algorithms when the run number reaches a certain threshold value.

6 Conclusion and Future Research

In this paper, a collection of aggregation algorithms was described and analyzed. These algorithms operate directly on compressed data sets in *MDW*s without the need to first decompress them. The algorithms are only applicable to the mapping-complete compression methods that are often used for compressing *MDW*s. A decision procedure is also given to select the most efficient algorithm based on aggregation request, available memory, as well as the data set parameters for a given aggregation request. The analysis and experimental results show that the proposed algorithms in this paper have better performance than the previous 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 algorithms for other operations, such as searching, cube, and other OLAP operations, on compressed *MDW*s. We are also working on algorithms for OLAP operations applicable to other kinds of compression methods other than mapping-complete compression methods.

ACKNOWLEDGMENTS

This work is supported in part by the Natural Science Foundation of China under Grant No. 69873014 and in part by the 973 Plan of China through Grant No. G1999032704.

REFERENCES

[1] J. Gray, S. Chaudhuri, A. Bosworth, 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, pp. 29-53, Jan. 1997.

[2] S. Yazdani and S. Wong, *Data Warehousing with Oracle.* Upper Saddle River, N.J.: Prentice-Hall, 1997.

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

[4] D. Chatziantonian and K. Ross, "Querying Multiple Features in Relational Databases," *Proc. 22nd Int'l Conf. Very Large Data Bases,* pp. 295-306, Sept. 1996.

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

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

[7] G. Colliat, "OLAP, Relational and Multidimensional Databases Systems," *SIGMOD Record,* vol. 25, no. 3, Sept. 1996.

[8] M.A. Bassiouni, "Data Compression in Scientific and Statistical Databases," *IEEE Trans. Software Eng.,* vol. 11, no. 10, pp. 1047-1058, Oct. 1985.

[9] M.A. Roth and S.J. Van Horn, "Database Compression," *SIGMOD RECORD,* vol. 22, no. 3, pp.19-29, Sept. 1993.

[10] Y. Zhao, P.M. Deshpande, and J.F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregations," *Proc. 1997 ACM-SIGMOD Conf. Management of Data,* pp. 159-170, May 1997.

[11] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys,* vol. 25, no. 2, pp. 73-169, June 1993.

[12] V. Harinarayan, A. Rajaraman, and J.D. Ullman, "Implementing Data Cube Efficienly," *Proc. 1996 ACM-SIGMOD Conf. Management of Data,* pp. 205-216, June 1996.

[13] H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman, "Index Selection for OLAP," *Proc. 13rd Int'l Conf. Data Eng.,* pp. 208-219, Apr. 1997.

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

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

[16] S. Agarwal, R. Agrawal, P.M. Deshpande, et al., "On the Computation of Aggregations," *Proc. 22nd Int'l Conf. Very Large Data Bases,* pp. 506-521, Sept. 1996.

[17] A. Shoshani, "Statistical Databases: Characteristics, Problems and Some Solutions," *Proc. Eighth Int'l Conf. Very Large Data Base,* pp. 208-222, Sept. 1982.

[18] M.C. Chen and L.P. McNamee, "On the Data Model and Access Method of Summary Data Management," *IEEE Trans. Knowledge and Data Eng.,* vol. 1, no. 4, pp. 519-529, Dec. 1989.

[19] J. Srivastava, J.S.E. Tan, and V.Y. Lum, "TBSAM: An Access Method for Efficient Processing of Statistical Queries," *IEEE Trans. Knowledge and Data Eng.,* vol. 1, no. 4, pp. 414-423, Dec. 1989.

[20] *Statistical and Scientific Databases.* A. Michalewicz, ed., 1992.

[21] S. Eggers and A. Shoshani, "Efficient Access of Compressed Data," *Proc. Sixth Int'l Conf. Very Large Data Bases,* pp. 205-211, Oct. 1980.

[22] J. Li, H.K. Wang, and D. Rotem, "Batched International Searching on Databases," *Proc. Third Int'l Conf. Data Eng.,* pp.18-24, Feb. 1987.

[23] J. Li, D. Rotem, and H.K. Wang, "A New Compression Method with Fast Searching on Databases," *Proc. 19th Int'l Conf. Very Large Data Bases,* pp. 311-318, Sept. 1987.

[24] J. Li and J. Srivastava, "Aggregation Algorithms for Very Large Compressed Data Warehouses," technique report, Harbin Inst. of Technology, FTP://210.76.60.241, 1999.

Jianzhong Li received the Bachelor degree in mathematics from the Heilongjiang University, Harbin, China, in 1975, the MS, and PhD degrees in computer science from the Harbin Institute of Technology, in 1982 and 1985, respectively. He has been on the faculty of the Department of Computer Science and Engineering at the Harbin Institute of Technology, since 1987 and is currently a professor and chairman of the Department of Computer Science and Engineering at the Harbin Institute of Technology. From 1985 to 1987, he was a researcher in the Information Research Group at Lawrence Berkeley National Laboratory, Berkeley, California He has also been a visiting professor at the University of Minnesota, Minneapolis, from 1991 to 1992. His research interests include parallel databases, data warehouses, data mining, parallel processing, database techniques for Web, and multimedia. He has authored three books, including *Parallel Database Systems*, *Principle of Database Systems*, and *Digital Library*, and published more than 120 technical papers in refereed journals and conference proceedings in the areas of databases, parallel processing, database techniques for Web, data mining, data warehouses, and multimedia. He has delivered a number of invited presentations and participated in panel discussions on these topics. His professional activities have included service on various program committees, and he has refereed papers for varied journals and proceedings. He is a member of the IEEE Computer Society and the ACM.

Jaideep Srivastava received the BTech degree in computer science from the Indian Institute of Technology, Kanpur, India, in 1983, the MS, and PhD degrees in computer science from the University of California, Berkeley, in 1985 and 1988, respectively. He has been on the faculty of the Department of Computer Science and Engineering of the University of Minnesota, Minneapolis, since 1988, and is currently a professor. He served as a research engineer with Uptron Digital Systems in Lucknow, India, in 1983. He has published more than 110 papers in refereed journals and conference proceedings in the areas of databases, parallel processing, artificial intelligence, and multimedia; and he has delivered a number of invited presentations and participated in panel discussions on these topics. His professional activities have included service on various program committees and he has refereed papers for varied journals and proceedings, for events sponsored by the US National Science Foundation. He is a senior member of the IEEE and a member of the IEEE Computer Society and the ACM.