ELSEVIER

Data & Knowledge Engineering 34 (2000) 21–38

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Range queries in dynamic OLAP data cubes

Weifa Liang [a,*], Hui Wang [b], Maria E. Orlowska [b]

[a] *Department of Computer Science, Australian National University, Canberra ACT 0200, Australia*

[b] *Department of Computer Science and Electrical Engineering, University of Queensland, St. Lucia Qld 4072, Australia*

Received 12 August 1999; received in revised form 14 February 2000; accepted 13 March 2000

Abstract

A range query applies an aggregation operation (e.g., SUM) over all selected cells of an OLAP data cube where the selection is specified by providing ranges of values for numeric dimensions. Range sum queries on data cubes are a powerful analysis tool. Many application domains require that data cubes are updated often and the information provided by analysis tools are current or "near current". Existing techniques for range sum queries on data cubes, however, can incur update costs in the order of the size of the data cube. Since the size of a data cube is exponential in the number of its dimensions, rebuilding the entire data cube can be very costly and is not realistic. To cope with this dynamic data cube problem, a new approach has been introduced recently, which achieves constant time per range sum query while constraining each update cost within $O(n^{d/2})$, where d is the number of dimensions of the data cube and n is the number of distinct values of the domain at each dimension. In this paper, we provide a new algorithm for the problem which requires $O(n^{1/3})$ time for each range sum query and $O(n^{d/3})$ time for each update. Our algorithm improves the update time by a factor of $O(n^{d/6})$ in contrast to the current one for the problem $O(n^{d/2})$. Like all existing techniques, our approach to answering range sum queries is also based on some precomputed auxiliary information (prefix sums) that is used to answer ad hoc queries at run time. Under both the product model and a new model introduced in this paper, the total cost for updates and range queries of the proposed algorithm is smallest compared with the cost by all known algorithms. Therefore our algorithm reduces the overall time complexity for range sum queries significantly.

Keywords: Range sum query; Data cube; Precomputation; OLAP; Multidimensional database; The update on-the-fly; Query algorithm

1. Introduction

Aggregation is a predominant operation in decision support database systems. On-Line Analytical Processing (OLAP) [3] often needs to summarize data at various levels of detail and on various combinations of attributes. An increasingly popular data model for OLAP applications is the multidimensional database (MDDB), also known as a data cube [6]. To build a data cube for a data warehouse, certain attributes are chosen to be *measure attributes*, i.e., the attributes whose values are of interest. Other attributes, say d of them, are referred to as *dimensions* or the

* Corresponding author.

E-mail addresses: wliang@cs.anu.edu.au (W. Liang), hwang@csee.uq.edu.au (H. Wang), maria@csee.uq.edu.au (M.E. Orlowska).

0169-023X/00/$ - see front matter
PII: S0169-023X(00)00007-0

functional attributes. The measure attributes of those tuples with the same dimension attribute values are added up into an aggregate value. Thus, an MDDB can be viewed as a d-dimensional array, indexed by the values of the d-dimensional attributes, whose cells contain the values of the measure attributes for the corresponding combination of dimension attributes. For example, consider a data cube maintained by an insurance company. Assume that the data cube has four dimensions CUSTOMER_AGE, YEAR, STATE and INSURANCE_TYPE, and one measure attribute SALES. Further, assume that the domain of CUSTOMER_AGE is between 1 and 100, of year is 1990–1999, of STATE is the 7 states and territories in Australia, and of INSURANCE_TYPE is $\{home, auto, health\}$. The data cube contains $100 \times 10 \times 7 \times 3$ cells, and each cell contains the total sales for the corresponding combination of CUSTOMER_AGE, YEAR, STATE, and INSURANCE_TYPE. Such a data cube provides the aggregate sales figures for all regions and year from 1990 to 1999.

We consider a class of queries over data cubes which are called *range queries* [9], that apply a given aggregation operation over selected cells where the selection is specified as contiguous ranges in the domains of some of the attributes. Such range queries are very useful tools in finding trends and in discovering relationships between attributes in data warehousing. One such example is the above insurance data cube: *find the total sales for customers with age from 35 to 50, in year 1994–1998 in ACT area with auto insurance*. One way to answer this query is to scan all involved cells in the data cube, and then add-up the values in the cells as the returning answer. However, this takes time which is proportional to the number of involved cells, while the number of cells in a data cube is exponential with its dimensions d. In an interactive exploration of a data cube, which is predominant for the OLAP applications, the response time is very crucial. It is imperative to have a system with fast response time. Our aim in this paper is to develop algorithms which reduce the response time of queries and updates significantly.

1.1. Related work

Since the introduction of the data cube model in [6], there has been considerable research in the database community on the development of algorithms for computing data cubes [1,4,13,15,18], for deciding what subsets of a data cube to precompute [7,8], for estimating the size of multidimensional aggregates [16], for indexing precomputed summaries [11,14], and for the maintenance of data cubes in the data warehouses [12]. However, the range query sum problem on the data cubes has not been received deserved attention until recently. Ho et al. [9] initialized the problem by presenting an elegant algorithm for it, based on the prefix sum approach. The essential idea of their method is to precompute prefix sums of cells in the data cube, which then can be used to answer ad hoc range queries at run-time. This method promises that any range sum query on a data cube can be answered in constant time. However, the updates to the prefix sums are proportional to the size of the data cube in the worst case when there is an update to a cell in the data cube. In some application where the data is static or rarely updated, this method is sufficient. There are, however, some applications for which the updates to the data cube happen quite often. For example, many corporations are interested in tracking down current sales data, for which new information may arrive on a daily or weekly basis. As competition increases globally, executives and managers demand that their analysis tools provide them with up-to-date information (current or near current result) without much

delay. For such applications, the prefix sum approach is not applicable, particularly for large data cubes having many dimensions, because the update time is in the order of the size of the data cube which grows exponentially with the number of dimensions d of the data cube. Thus, it leaves us a challenging problem, that is whether there is an efficient algorithm which has a reasonable update time and small query time. Geffner et al. [5] first considered this *dynamic data cube* problem by presenting a solution. Their algorithm for the range sum query takes $O(n^{d/2})$ time for an update and constant time for a query, assuming the data cube has d-dimensions and each dimension is of size n. Chan and Ioannidis [2] proposed a new class of cube representations called *Hierarchical Cubes*, which is based on two orthogonal dimensions. They have shown that a particular cube design called the *Hierarchical Band Cube* has a significantly better query and update trade-off than that of the algorithm due to [5]. But the index mapping from a high-level "abstract" cube to a lower-level "concrete" cube is too complicated. In addition to the above approaches, which provide precise answers to range sum queries, a method that provides approximate answers for high-dimensional, sparse data cube has recently been proposed as well [17].

It should be mentioned that Ho et al. [10] also presented efficient techniques for partial sum queries where queries are on arbitrary subsets (not necessarily contiguous) of the measure attributes. They mapped the partial-sum problem to a covering problem in the theory of error-correcting codes, applied some known covering codes to the problem, and devised a new covering code tailored for the partial-sum query problem that offers the best space and time trade-off. Although a range sum query problem can be viewed as a special case of the partial-sum query problem, the techniques specified for range sum queries usually take advantage of the contiguous ranges of the selection and have a much better performance.

As Ho et al. [9] pointed out, their techniques can also be applied to any binary operator $\oplus$ for which there is an *inverse* binary operator $\ominus$ such that $a \oplus b \ominus b = a$ for any a and b in the domain. Examples for such applications are `COUNT`, `AVERAGE`, `ROLLING SUM`, `ROLLING AVERAGE`, etc.

1.2. Contributions

In this paper, we present a new technique called a *double relative prefix sum* approach to speed-up range sum queries in data cubes, which achieves $O(n^{1/3})$ time per query and $O(n^{d/3})$ time per update, where n is the number of distinct values of the domain at each dimension. Compared with the existing algorithms, our algorithm improves the update time by a factor of $O(n^{d/6})$ while the query time is kept within $O(n^{1/3})$ time.

The remainder of the paper is organized as follows. In Section 2, we introduce the model for the range sum problem [9], and present the optimization measure of the problem under different (time) cost models. We also review some existing approaches for the problem. In Section 3, we present a new technique for the range sum query problem on dynamic data cubes which aims to minimize the total cost for both queries and updates under the two cost models. In doing so, we first describe several data structures used to accommodate our task. We then devise algorithms for processing range sum queries and updates on these data structures. In Section 4, we present a general strategy for dealing with range sum queries on a set of data cubes in a dynamic data warehouse environment. We conclude our discussion in Section 5.

2. Preliminaries

2.1. The model

Following [5,9], we use the same model for the range sum problem. Assume that the data cube has one measure attribute and d-dimension. Let $D = \{1, 2, \ldots, d\}$ denote the set of dimensions. Each dimension has a size n_i, which represents the number of distinct values in the domain at that dimension. This size is static and known *a priori*. Thus, a d-dimensional data cube can be represented by a d-dimensional array A of size $n_1 \times n_2 \times \cdots \times n_d$, where $n_j \geqslant 2$, $j \in D$. Each entry in array A is called a *cell*, and each cell contains the aggregate value of the measure attribute corresponding to a given point in the d-dimensional space formed by the d-dimensions. Without loss of generality and for simplicity, we assume that A has a starting index 0 at each dimension, and each dimension has the same size n. This allows us to present subsequent formulae more concisely. Thus, the total size of array A is $N = n^d$. Note that the same assumption has been used in [5].

The range sum query problem is to find the sum of the values in cells that fall within the specified range (usually a hyper-rectangle, hereafter called a d-dimensional "box" within the d-dimensional array A). For example, in the two-dimensional case, given a measure attribute `SALES`, the dimensions `CUSTOMER_AGE` and `YEAR`, the cell at $A[37, 8]$ contains the total sales to age 37 years old customers in 1997. A range sum query asking for the total sales to 37 years old customers from year 1997 to 1999 would be answered by summing the cells $A[37, 8]$, $A[37, 9]$, and $A[37, 10]$. In the following, we also refer to the range sum query problem as the range query problem.

2.2. Existing approaches

There is a naive method of answering the range queries on a data cube. Assume that the data cube stored by an array A has been built. An arbitrary range query on A takes $O(n^d)$ time in the worst case because all the cells in A must be added up in order to answer the query. This approach, however, is very simple in dealing with an update which takes only $O(1)$ time for each cell update. Therefore, it is ideal for a data cube frequently updated but infrequently queried. This assumption, however, may not be realistic.

In order to improve the range query performance, Ho et al. [9] presented an elegant algorithm which is called *the prefix sum method*. In the prefix sum method, besides A, another d-dimensional array P of size $N = n_1 \times n_2 \times \cdots \times n_d$ is introduced, which has the same size as array A. P is used to store various precomputed prefix sums of A. Each cell indexed by $(x_1, x_2, \ldots, x_d)$ in P contains the sum of all cells up to and including itself in array A. Thus, the sum of the entire array A is found in the last cell of P. In other words, we will precompute, for all $0 \leqslant x_i < n_i$ and $i \in D$,

$$P[x_1, x_2, \ldots, x_d] = \mathrm{Sum}(A[0, 0, \ldots, 0] : A[x_1, x_2, \ldots, x_d]) = \sum_{i_1=0}^{x_1} \sum_{i_2=0}^{x_2} \cdots \sum_{i_d=0}^{x_d} A[i_1, i_2, \ldots, i_d].$$

For example, when $d = 2$, we precompute, for all $0 \leqslant x < n_1$ and $0 \leqslant y < n_2$,

$$P[x, y] = \sum_{i=0}^{x} \sum_{j=0}^{y} A[i, j].$$

Using P, any range sum query on d-dimensions can be answered through constant ($2^d = O(1)$) cells look-ups, which has been proved by the following lemma. For notational convenience, let $P[x_1, x_2, \ldots, x_d] = 0$ if $x_j = -1$ for some $j \in D$.

Lemma 1 [9]. *For all $j \in D$, let*

$$s(j) = \begin{cases} 1 & \text{if } x_j = h_j, \\ -1 & \text{if } x_j = l_j - 1. \end{cases}$$

Then, for all $j \in D$,

$$\text{Sum}(A[l_1, l_2, \ldots, l_d] : A[h_1, h_2, \ldots, h_d]) = \sum_{x_j \in \{l_j - 1, h_j\}} \{(\Pi_{i=1}^{d} s(i)) P[x_1, x_2, \ldots, x_d]\}.$$

Fig. 1 illustrates the basic idea of the prefix sum approach when $d = 2$. The sum corresponding to a range query's region can be determined by adding and subtracting the sums of various other regions until the interesting region is extracted. Thus, the prefix sum approach has reduced the range sum query problem to the problem of reading a single individual cell in array P which takes constant time.

The prefix sum approach is very powerful. It provides range sum queries in constant time, regardless of the size of the data cube. On the other hand, the update cost is very expensive and is proportional to the entire array size. In the worst case, the update cost is $O(n^d)$. Thus, the entire system performance (partially contributed by the update times) is deteriorated heavily when the queries and updates arise equally likely. The update costs result from the dependencies in the data that allow the approach to work. The values of cells in P are cumulative. Therefore, the prefix sum approach is not suitable to the case where the cell values in a data cube need to be updated frequently.

To reduce the update time without sacrificing the constant query time, Geffner [5] presented an approach called *relative prefix sum method* to reduce the update time significantly. Basically, the relative prefix method follows the same idea as [9], i.e., adds and subtracts region sums to obtain the sum of the interesting region query. The difference is that this latter method makes use of new data structures that provide constant time for range queries while controlling the cumulative updates by creating boundaries that limit cascading updates to distinguished cells. The relative prefix sum method uses two arrays as its data structures. One is *overlay array*, the other is *relative prefix array RP*.

An overlay partitions array A into fixed size regions is called *overlay boxes*. There is a corresponding *box cell* in the overlay array which stores information regarding the sums of regions of

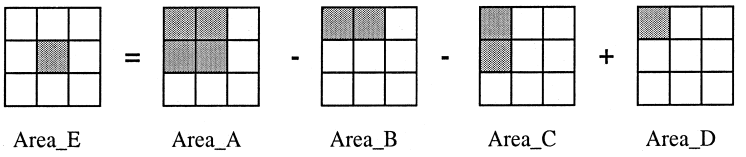

Fig. 1. A geometric illustration for the two-dimensional case: Sum(Area_E) = Sum(Area_A) − Sum(Area_B) − Sum(Area_C)+ Sum(Area_D).

array A which precede it. RP is an array of the same size as array A; it contains relative prefix sums within regions defined by the overlay boxes. Using the two arrays, the prefix sum of each cell in A can be constructed in constant time and can be done "on-the-fly". Compared with the prefix sum approach, it improves the update time by an $O(n^{d/2})$ factor, but it needs more space to store the segmented prefix sums. In the following the more details about the relative prefix sum method will be given, because our algorithm will use it as a subroutine.

Let r be the size of each dimension in an overlay box and n be divisible by r. Thus, each overlay box "covers" r^d cells in A, and array A is partitioned into $(n/r)^d$ disjoint overlay boxes.

The relative prefix array RP has the same size as array A. It is partitioned into regions of cells that correspond to overlay boxes. Each overlay box covers a region, and therefore, each region contains r^d cells. Also, each cell in a region contains the prefix sum that is relative to the area enclosed by the overlay box. Formally speaking, given a cell $RP[x_1, x_2, \ldots, x_d]$ which is covered by an overlay box B, then,

$$RP[x_1, x_2, \ldots, x_d] = \mathrm{Sum}(A[a_1, a_2, \ldots, a_d] : A[x_1, x_2, \ldots, x_d]),$$

where $(a_1, a_2, \ldots, a_d)$ is the lowest index of a cell at each dimension in the overlay box B.

Let $(a_1, a_2, \ldots, a_d)$ be the lowest index of a cell at each dimension in an overlay box. Define the *volume* V of the overlay box

$$V = \sum_{i_1=0}^{a_1-1} \sum_{i_2=0}^{a_2-1} \cdots \sum_{i_d=0}^{a_d-1} A[i_1, i_2, \ldots, i_d].$$

In other words, V is the prefix sum of all cells preceding $A[a_1, a_2, \ldots, a_d]$ in array A.

In the overlay array, each box cell contains $dr^{d-1} + 1$ elements (values). These elements include the volume V of the overlay box and the other dr^{d-1} *border values* [1] of the box. The border values in an overlay box provide sums of regions outside the box, which will be used when computing the prefix sum of the cells in the box in array A. The border values are equal to the sum of the associated shaded regions of array A (see Fig. 2). For example, in the two-dimensional case, X_1 is the sum of all cells in the column above the cell containing X_1. Border value X_2 is the sum of all cells in the column above its cell, plus the cells above X_1, and X_i is the sum of X_{i-1} and all cells above X_i in the column. Similarly, Y_1, Y_2, and Y_k can be defined on the corresponding rows. In a d-dimensional array, the border values of an overlay box can be computed as dimension by dimension. Formally, the border value contained in a cell indexed by $(x_1, x_2, \ldots, x_d)$ is $\mathrm{Sum}(A[0, 0, \ldots, 0] : A[x_1, x_2, \ldots, x_d]) - RP[x_1, x_2, \ldots, x_d] - V$, where V is the volume of the overlay box and $RP[x_1, x_2, \ldots, x_d]$ is the value of this cell in array RP.

Having the data structures above, we first consider an update to a cell in array A. To maintain the data structures, we first update the region in which the cell located, i.e., update the affected cells in RP in this region, which takes $O(r^d)$ time. We then update the affected boxes in the overlay array, i.e., we need to update the volumes and the border values of those affected boxes, which takes $O((n/r)^d + n/rdr^{d-1})$ time, where $O(n/r)^d$ is the time to update the volumes of boxes involved, and $O(n/r)dr^{d-1}$ is the time to update the border values of the n/r boxes involved at each

[1] Here we have d more border values than in [5].

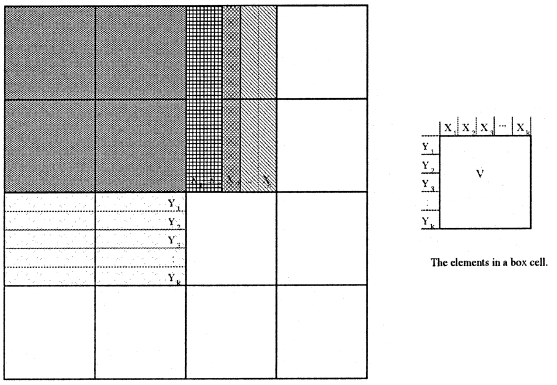

Fig. 2. A geometric illustration for the two-dimensional case.

dimension. To minimize the update time, r is chosen to be $\sqrt{n}$. That is, the update time for every update is $O(n^{d/2})$ in the worst case.

We then consider a range sum query on the data cube A. As mentioned in Section 2, each range sum query needs calculating 2^d region sums as illustrated in Fig. 1. The calculation of each region sum requires adding the volume of the region (overlay box), d border values, and the relative prefix sum which comes from RP. Thus, each range sum query can be answered in constant time.

We finally consider the space requirement for this approach. The space used by the overlay array is $(n/r)^d(dr^{d-1}+1) = dn^d/r + (n/r)^d$ because there are $(n/r)^d$ overlay boxes in the overlay array and each of them contains $dr^{d-1}+1$ elements. Thus, when $r = \sqrt{n}$, the ratio of the space used between the relative prefix sum approach and the naive approach is

$$\frac{2n^d + dn^d/\sqrt{n} + (n/\sqrt{n})^d}{n^d} = 2 + d/\sqrt{n} + 1/n^{d/2} \approx 2 + d/\sqrt{n},$$

where $2n^d$ is the space used for A and RP, and $dn^d/\sqrt{n} + (n/\sqrt{n})^d$ is the space used for the overlay array.

2.3. Cost models

For a given data cube, assume that each range query takes time t_q and each update takes time t_u. Both t_u and t_q are the time in the worst case. We further assume that the range sum query and the update operations on the data cube are mutually exclusive. That is, the data cube is not available for user queries during its update period. Similarly, the update operation cannot proceed when a range query operation is being evaluated. In [5] they take the product of query time and update time $O(t_u \cdot t_q)$ as an optimization objective. When $t_q = O(1)$ is fixed, they present an algorithm which achieves $t_u = O(n^{d/2})$, where d is the number of dimensions and n is the size of each dimension. This cost model is very useful when dealing with a data cube that has lot of queries but few updates. Clearly, this approach is inappropriate for dealing with the situation where queries and updates arise equally likely or periodically. For example, assume that there is a data cube for which there are 10 queries (2 queries per working day) and 1 update (weekend) every week on average. From this example, we can see, during a given time window (daily, weekly, or monthly),

the ratio between the number of range queries and the number of updates to a data cube is approximately fixed. Thus, in order to reduce the total time of range sum queries and updates, it needs to minimize both query and update times. In the following, we introduce a new model which we believe is more accurate in modeling this latter case than the existing ones.

2.3.1. A new cost model

For a given time window, assume that the average numbers of range sum queries and updates to a data cube are known in advance. Thus, if there are n_q queries and n_u updates during the given time window, then, the total time used for both range queries and updates is $n_q t_q + n_u t_u$. Let $c = n_q/n_u$ which is the *ratio of query and update*. Usually, c is fixed. For the above example, $c = 10$ if a week is assumed to be the given time window. Here our objective is to minimize $n_q t_q + n_u t_u = n_u(ct_q + t_u)$ since $n_q = cn_u$.

Under this new cost model, if the ratio between the updates and the queries is fixed, the update time given by Geffner et al. [5] is not the best. In this paper, we will devise an algorithm based on our new cost model, which makes the update time much smaller than the currently best one. The improvement of the update time is at the expense of increasing the query time. But the increase in the query time is insignificant in our case. As a result, the total time spent for range queries and updates by our algorithm is much smaller than that of existing algorithms, under both the product model and the new cost model.

3. Double relative prefix sum approach

In this section, we present a technique called *Double relative prefix sum approach* for the range sum query problem. Our approach is inspired by the algorithm in [5], which limits the updating of cumulative sums. The rest of this section is organized as follows. We first define the data structures that will be used in our algorithms. We then present algorithms for updating and range query operations on the data structures. We finally analyze the time and space complexity of the proposed algorithms required.

3.1. Data structures

Assume that a data cube is stored by a d-dimensional array A. For example, Fig. 3 shows a two-dimensional data cube A. We will use it as example later to illustrate the steps of our algorithms. The proposed algorithm makes use of three data structures: *relative prefix array RP*, *relative overlay array RO* and the *block prefix array BP*, respectively. All of the three arrays are d-dimensional arrays which are defined below.

3.1.1. Relative prefix array

The relative prefix array RP has the same size as array A. The cells in A are partitioned into disjoint *overlay boxes*. A cell in an overlay box contains the prefix sum of values of the cells preceding it and includes itself within that box. Assume that r is the size of each dimension of an overlay box, then there are $\lceil n/r \rceil^d$ boxes after the partition of A, and all prefix sums in each box can be computed in $O(r^d)$ time. Without loss of generality, we assume n is divisible by r. Let $(a_1, a_2, \ldots, a_d)$ be the lowest index of each dimension in an overlay box and $(x_1, x_2, \ldots, x_d)$ be the index of a cell in the box. Then, the value of the cell in RP is

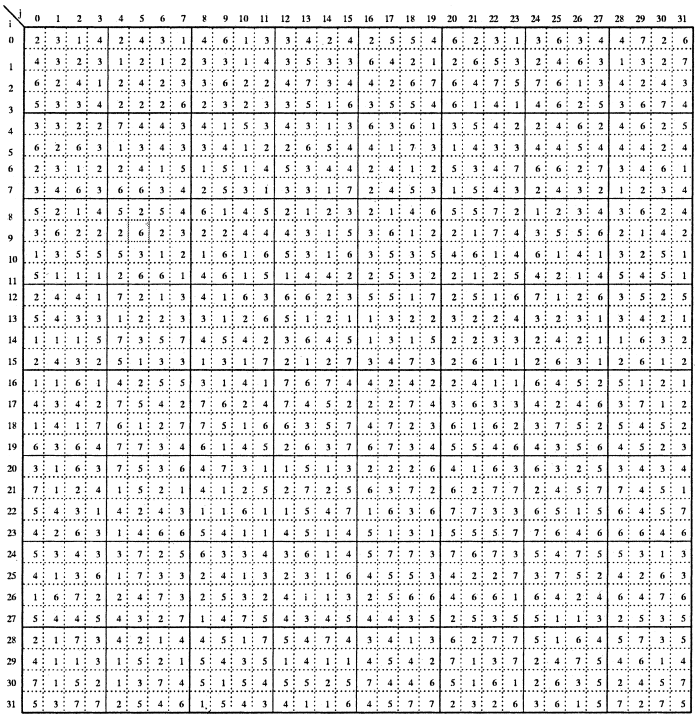

Fig. 3. A two-dimensional data cube array A.

$$RP[x_1, x_2, \ldots, x_d] = \mathrm{Sum}(A[a_1, a_2, \ldots, a_d] : A[x_1, x_2, \ldots, x_d]).$$

For the above example, Fig. 4 shows its relative prefix array RP (incomplete), where $n = 32$ and $r = 4$. Each solid box in Fig. 4 is an overlay box. Now, let us compute $RP[6, 9]$. The lowest index of the cell at each dimension of the overlay box which contains the cell is $(4, 8)$. Therefore, $RP[6, 9] = \sum_{i=4}^{6} \sum_{j=8}^{9} A[i, j] = 18$.

3.1.2. Relative overlay array

We now consider the relative overlay array RO. There is a corresponding *box cell* in RO for every overlay box obtained through the partition of array A. Thus, array RO consists of $(n/r)^d$ box cells. RO can be thought as a compressed representation of A. We further partition the box cells in RO into disjoint *block boxes*. Each dimension of a block box has the same size rr_1. Thus, there are $(n/rr_1)^d$ block boxes for array A with an assumption that $n \bmod rr_1 = 0$.

Each box cell in RO contains $dr^{d-1} + 1$ elements, in which the dr^{d-1} elements are the border values of the overlay box relative to a block box in which the box cell is included, and another element V_{ro} is the *relative volume* of the overlay box (i.e., the relative prefix sum within its block box), which is defined as follows. Let $(x_1, x_2, \ldots, x_d)$ be the lowest index of a cell at each dimension in an overlay box and the overlay box is within a block box where $(a_1, a_2, \ldots, a_d)$ is the lowest index of the cell at each dimension in the block box. Then,

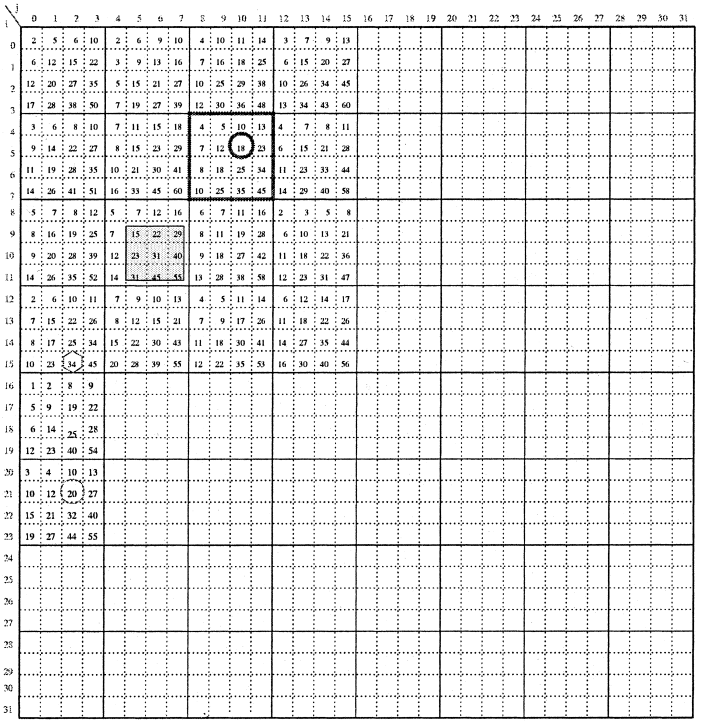

Fig. 4. The relative prefix array *RP* of the data cube *A* (partially).

$$V_{ro} = \mathrm{Sum}(A[a_1, a_2, \ldots, a_d] : A[x_1 - 1, x_2 - 1, \ldots, x_d - 1]) = \sum_{i_1=a_1}^{x_1-1} \sum_{i_2=a_2}^{x_2-1} \cdots \sum_{i_d=a_d}^{x_d-1} A[i_1, i_2, \ldots, i_d],$$

i.e., V_{ro} is the relative prefix sum of those cells which are preceding $A[x_1, x_2, \ldots, x_d]$ in array A and are covered by the block box.

The border values of an overlay box are defined but are bounded in a block box to which the overlay box belongs. The border values are the sums of those regions which are outside the box but within the block box. For the above example, there are $(n/rr_1)^2 = 4$ block boxes when $r = 4$ and $r_1 = 4$. Fig. 5 shows an incomplete array *RO* for our example. Consider an overlay box whose lowest index of a cell at each dimension is $A[8, 8]$. We now compute the border values and relative volume of this box within its block box (here is block box 1). Note that the lowest index of a cell at each dimension in the block box 1 is $(0, 0)$. Then, $V_{ro} = \sum_{i=0}^{8-1} \sum_{j=0}^{8-1} A[i, j] = 200$. Now we consider the border values of this box cell. The array has two dimensions (column and row). The border values at the column dimension are X_1, X_2, X_3 and X_4.

$$X_1 = \sum_{i=0}^{8-1} \sum_{j=8}^{8} A[i, j] = 22,$$

$$X_2 = \sum_{i=0}^{8-1} \sum_{j=8}^{9} A[i, j] = X_1 + \sum_{i=0}^{8-1} \sum_{j=9}^{9} A[i, j] = 22 + 33 = 55,$$

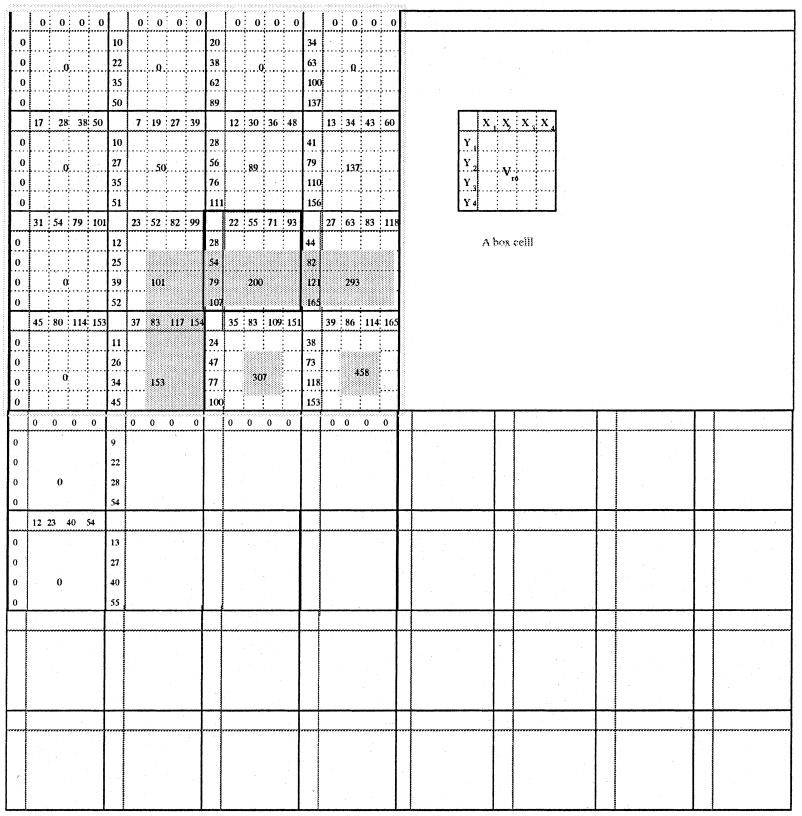

Fig. 5. The relative overlay array *RO* of the data cube *A* (partially).

$$X_3 = X_2 + \sum_{i=0}^{8-1} \sum_{j=10}^{10} A[i,j] = 55 + 16 = 71$$

and

$$X_4 = X_3 + \sum_{i=0}^{8-1} \sum_{j=11}^{11} A[i,j] = 93.$$

The border values at the row dimension Y_1, Y_2, Y_3 and Y_4 can be computed similarly.

3.1.3. Block prefix array

As described above, the box cells of *RO* are partitioned into block boxes, and rr_1 is the size of each dimension of a block box, i.e., each block box covers r_1^d box cells and "covers" $(rr_1)^d$ cells in array *A* indirectly. Thus, an array *BP* is used for this purpose, which contains $(n/rr_1)^d$ cells, and each cell in *BP* contains the volume *V* of the corresponding block box. Let $(a_1, a_2, \ldots, a_d)$ be the

lowest index of a cell at each dimension in a block box, then, the volume V of the block box is defined as

$$V = \mathrm{Sum}(A[0,0,\ldots,0] : A[a_1 - 1, a_2 - 1, \ldots, a_d - 1]).$$

For our example, the block prefix array BP for the two-dimensional array A is as follows. Consider a block box whose cells' lowest index is $(16,16)$. The corresponding cell in BP of this block box is $BP[1,1] = \sum_{i=0}^{16-1} \sum_{j=0}^{16-1} A[i,j] = 776$.

3.1.4. Space requirement

The space required for the data structures above is analyzed as follows. The space for A and RP is $2n^d$; the space for array RO is $(n/r)^d(dr^{d-1} + 1) = dn^d/r + (n/r)^d$; and the space for array BP is $(n/rr_1)^d$. Thus, the ratio of the space required between the data structures above and the naive approach is

$$\frac{2n^d + dn^d/r + (n/r)^d + (n/rr_1)^d}{n^d} = 2 + d/r + 1/r^d + 1/(rr_1)^d = 2 + d/n^{1/3} + 1/n^{d/3} + 1/n^{2d/3} \approx 2 + d/n^{1/3},$$

when $r = r_1 = \lceil n^{1/3} \rceil$.

3.2. Algorithms

Having built the data structures above, we now present algorithms to implement update and range query operations on the data structures.

3.2.1. Update cost

We now consider how each update affects the data structures. Assume that a cell in array A has been updated. To response to this update, the data structures need to be maintained. We proceed as follows.

First, we update some affected cells in an overlay box in array RP in which the updated cell is contained. In other words, every cell in the overlay box whose index is larger than the index of the updated cell will be updated. This takes $\mathrm{O}(r^d)$ time in the worst case. For our example, consider cell $A[9,5]$ is updated, then, all the cells in the shadow area in Fig. 4 need to be updated.

Then, we update the contents of box cells in RO covered by a block box in which the updated box cell is contained, i.e., we update the border values and relative volumes of some box cells within the block box in array RO. This takes $\mathrm{O}(r_1^d + r_1(dr^{d-1} + 1)) = \mathrm{O}(dr_1 r^{d-1} + r_1^d)$ time because under the coverage of the block box, the relative volumes of r_1^d box cells may need to be updated, and the border values of the dr_1 box cells may also need to be updated. For our example, consider cell $A[9,5]$ is updated, then, the content (border values and/or relative volumes) of all the box cells in the shadow areas in Fig. 5 need to be updated.

Finally, we update the cells in block prefix array BP which takes $\mathrm{O}(n/rr_1)^d$ time because BP contains $\mathrm{O}(n/rr_1)^d$ cells. For our example, consider cell $A[9,5]$ is updated, then, every block box whose cells' lowest index is larger than the lowest index of cells within the block box covering $A[9,5]$ must be updated, i.e., the cells in the shadow area of Fig. 6 need to be updated.

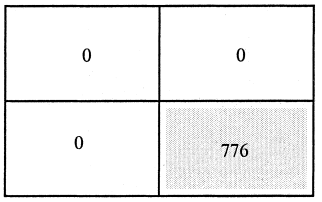

Fig. 6. The block prefix array *RO* of the data cube *A*.

Thus, the total time for the maintenance of data structures due to an update is $O(r^d + dr_1r^{d-1} + r_1^d + (n/rr_1)^d)$. To minimize the total time of an update, we set $r = r_1 = n/rr_1$, i.e., $r = r_1 = \lceil n^{1/3} \rceil$. So, the time for each update is $O(n^{d/3})$.

3.2.2. Query cost

Now we consider the range sum query cost. Assume that a user has the following range sum query: $\text{Sum}(A[l_1 : h_1, l_2 : h_2, \ldots, l_d : h_d])$ over a data cube A with schema $(A_1, A_2, \ldots, A_d, M)$ where A_i is a dimensional attribute and M is a measure attribute of data cube A, $1 \leqslant i \leqslant d$. This query can be written in a SQL-like language as follows.

SELECT	SUM($A.M$)
FROM	d-dimensional array A
WHERE	$(l_1 \leqslant A.A_1 \leqslant h_1)$ AND $(l_2 \leqslant A.A_2 \leqslant h_2)$AND ... AND $(l_d \leqslant A.A_d \leqslant h_d)$

The above statement can be translated into finding the prefix sums of 2^d cells of A by Lemma 1. Thus, the problem now reduces to find the prefix sum of a cell in array A. For a given cell indexed by $(x_1, x_2, \ldots, x_d)$, our objective is to compute its prefix sum in A. In doing so, we first compute the relative prefix sum of the cell in a block box B (the cell is covered by an overlay box and the overlay box is then covered by a block box B). We have the following lemma.

Lemma 2. *The relative prefix sum of a cell related to its block box can be found in* $O(1)$ *time using the above data structures.*

Proof. A box cell in *RO* is first located, in which the cell is contained, by running the algorithm [5]. Then, the relative volume V_{ro} of the box cell and its d border values can be found by retrieving array *RO*. So, the relative prefix sum *RPS* of the cell is

$$RPS[x_1, x_2, \ldots, x_d] = V_{ro} + \sum_{j=1}^{d} XY_j + RP[x_1, x_2, \ldots, x_d],$$

where XY_j is the relative border value of the box cell at dimension j within the block box, $1 \leqslant j \leqslant d$. Since V_{ro} and XY_j can be obtained through retrieving the box cell in *RO* and $RP[x_1, x_2, \ldots, x_d]$ can be obtained from array *RP*. Also, the number of dimensions d in a data cube is fixed. Thus, finding the relative prefix sum of a cell in a block box can be done in constant time.

For our example, consider the relative prefix sum $RPS[10,10]$ of a cell indexed by $(10,10)$. First, from the index of this cell, we can find the overlay box in which the cell is contained. The lowest index of cells at each dimension of the overlay box is $(8,8)$ whose relative volume is $V_{ro} = 200$ and whose border values to this cell is $X_3 = 71$ and $Y_3 = 79$. So, $RPS[10,10] = V_{ro} + X_3 + Y_3 + RP[10,10] = 200 + 71 + 79 + 27 = 377$.

We then compute the prefix sum of a cell indexed by $(x_1, x_2, \ldots, x_d)$ in A only because each range sum query can be reduced to find 2^d of prefix sums of the cells in array A. To simplify the problem, we assume that the border values of each block box also exist, which do not exist actually. Later we remove this assumption by showing how to compute those required border values on-the-fly. Let B be the block box which covers the cell indexed by $(x_1, x_2, \ldots, x_d)$. Also, let $XY_1, XY_2, \ldots, XY_d$ be the d border values of B associated with the cell indexed by $(x_1, x_2, \ldots, x_d)$, where XY_j is the border value at j dimension, $1 \leqslant j \leqslant d$. The volume V of B can be found from array BP. Then, the prefix sum of the cell in array A is

$$P[x_1, x_2, \ldots, x_d] = V + \sum_{j=1}^{d} XY_j + RPS[x_1, x_2, \ldots, x_d].$$

Thus, the prefix sum of the cell can be computed in constant time. However, the border values XY_j of the block box B used here have not been precomputed, $1 \leqslant j \leqslant d$. So, we need to compute them on-the-fly.

Now we compute XY_j at dimension j. Assume that there are k block boxes $B_1, B_2, \ldots, B_k$ preceding B at this dimension. Having a cell indexed by $(x_1, x_2, \ldots, x_d)$, let OB be the overlay box containing the cell, then, the border cell of OB indexed by $(b_1, b_2, \ldots, b_d)$ corresponding the cell and the corresponding border value XY_j can be derived immediately. Similarly, the cell indexed by $(b_1 - (k-l)rr_1 - 1, b_2 - (k-l)rr_1 - 1, \ldots, b_d - (k-l)rr_1 - 1)$ in B_l (a border value cell in B_l) can also be derived, $1 \leqslant l \leqslant k$. The relative prefix sum $RPS[b_1 - (k-l)rr_1 - 1, b_2 - (k-l)rr_1 - 1, \ldots, b_d - (k-l)rr_1 - 1]$ of the cell indexed by $(b_1 - (k-l)rr_1 - 1, b_2 - (k-l)rr_1 - 1, \ldots, b_d - (k-l)rr_1 - 1)$ within B_l can be found in constant time by Lemma 2. Thus, the border value XY_j of B at dimension j is $XY_j = \sum_{l=1}^{k} RPS[b_1 - (k-l)rr_1 - 1, b_2 - (k-l)rr_1 - 1, \ldots, b_d - (k-l)rr_1 - 1]$, where $1 \leqslant k \leqslant \lceil n/rr_1 \rceil$. When $r = r_1 = \lceil n^{1/3} \rceil$, XY_j can be found in $O(n^{1/3})$ time. Therefore, each range query can be answered in $O(n^{1/3})$ time. Fig. 7 gives an explanation of computing the border values of a block box, where "$*$" represents the border cells in different block boxes, and XY_j is equal to the summation of all cells in array A in the shadow area. For our two-dimensional array example, consider the prefix sum of a cell indexed by $(21,2)$ in array A. By our algorithm, we first find in which overlay box this cell is contained. It is included in an overlay box whose lowest index at cell $(20,0)$, and the overlay box is covered by a block box whose lowest cell index is $(16,0)$, and the volume V of the block box is 0 ($V = BP[1,0] = 0$). We then compute the relative prefix sum $RPS[21,2]$ of the cell in the block box, and $RPS[21,2] = V_{ro} + X_3 + Y_2 + RP[21,2] = 0 + 40 + 0 + 20 = 60$.

We finally find the border values of the block box related to this cell. For this cell, there are two corresponding border cells whose indexes are $(16,2)$ and $(21,0)$, respectively. For the column dimension, there is one block box B_1 preceding the current block box, and the corresponding border cell is $(16-1,2)$ within B_1, so, $RPS[15,2] = V_{ro} + X_3 + Y_4 + RP[15,2] = 0 + 114 + 0 + 34 =$

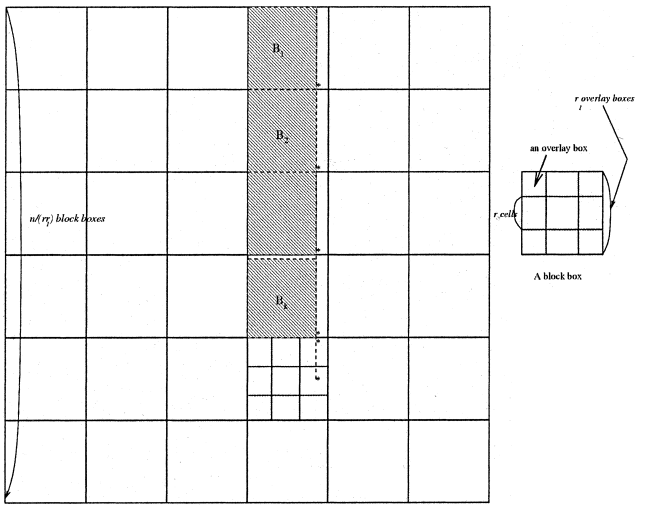

Fig. 7. An illustration of computing a border value XY_j of a block box.

148. So, the border value of the block box corresponding the cell in the column dimension is $XY_1 = RPS[15, 2] = 148$. For the row dimension, there is no block box preceding the current block box. So, the border value of the block box corresponding to the cell in this dimension is $XY_2 = 0$. Thus, the prefix sum of the cell in array A is $P[21, 2] = V + XY_1 + XY_2 + RPS[21, 2] = 0 + 148 + 0 + 60 = 208$. Thus, we have the following theorem.

Theorem 1. *There is an algorithm for the range sum problem on a dynamic OLAP data cube. The query time is* $O(n^{1/3})$ *and the update time is* $O(n^{d/3})$, *where n is the size of each dimension and d is the number of dimensions of the data cube.*

3.3. Total cost measure

Now we compare our algorithms with other previously known algorithms for the range sum query problem under the two cost models.

Under the product cost model (query time times update time), the cost by our algorithm is $O(n^{1/3}n^{d/3}) = O(n^{(d+1)/3})$, while the cost of the algorithm in [5] is $O(n^{d/2})$, and the cost of the algorithm in [9] and the naive approach is $O(n^d)$.

Under the new cost model, assume that there are K queries and updates and the ratio between the number of queries and the number of updates is c, ($\geqslant 1$). Let n_u be the number of updates among the K operations, then, $(c + 1)n_u = K$. The cost of the naive algorithm is $O(cn_u \cdot n^d + n_u) = O(Kn^d)$. The cost of the prefix sum approach is $O(cn_u + n_u n^d) = O(Kn^d)$. The cost of the relative prefix sum approach is $O(cn_u + n_u n^{d/2}) = O(Kn^{d/2})$. The cost of the double relative prefix sum approach is $O(cn_u \cdot n^{1/3} + n_u n^{d/3}) = O(c/(c + 1)\ Kn^{1/3} + 1/(c + 1)Kn^{d/3}) = O(Kn^{d/3})$.

Clearly, the performance of our algorithm outperforms that of all existent algorithms under the both cost models. Therefore, it reduces the overall cost of the problem. Compared with the relative prefix sum approach in [5], our algorithm, however, needs more space to store intermediate

Algorithm name	query time	update time	Time for q queries and p updates	S_p/n^d
Naive	$O(n^d)$	$O(1)$	$O(p+qn^d)$	1
Prefix sum	$O(1)$	$O(n^d)$	$O(pn^d+q)$	2
Relative prefix sum	$O(1)$	$O(n^{d/2})$	$O(pn^{d/2}+q)$	$2+d/\sqrt{n}$
Double relative prefix sum	$O(n^{1/3})$	$O(n^{d/3})$	$O(pn^{d/3}+qn^{1/3})$	$2+d/n^{1/3}$
Hierarch. Band Cube	N/A	N/A	N/A	N/A

Fig. 8. Time and space requirements among different algorithms.

results, but the increase in space is insignificant. In summary, the following table lists the time and space requirements of different algorithms, where S_p is the space used by the corresponding algorithm and n^d is the space used by the naive approach. (See Fig. 8)

4. Range sum queries on multiple data cubes

In this section, we discuss the range sum queries on different data cubes in a data warehouse. That is, how to maintain all the data cubes systematically so that the entire system performance can be improved.

Let $DW = \{DB_1, DB_2, \ldots, DB_w\}$ where DW represents the data warehouse, and DB_i is a data cube in the data warehouse, $1 \leqslant i \leqslant w$. Given a time window, let K_i be the number of queries and updates to DB_i, let $t_q^{(i)}$ and $t_u^{(i)}$ be the range query and update time to DB_i and c_i be the ratio between the number of queries and updates to DB_i, $1 \leqslant i \leqslant w$. Then, the update frequency f_i to DB_i is $f_i = (K_i/(c_i+1))/\sum_{j=1}^{w} K_j$ and the query frequency q_i to DB_i is $q_i = ((c_iK_i)/(c_i+1))/\sum_{j=1}^{w} K_j$, $1 \leqslant i \leqslant w$, and $\sum_{j=1}^{w}(f_i+q_i) = 1$. The total time for processing all the range sum queries and update operations on the data cubes in DW is $\sum_{j=1}^{w}(f_jK_jt_u^{(j)} + q_jK_jt_q^{(j)})$, for the given time window.

Following the discussions in previous sections, we have the following general policy to deal with range queries. For a given data cube $DB_i \in DW$, (i) if the data in it is rarely updated and the size of the data cube is not too large (it can be stored in main memory), the naive approach will be used for range sum queries to it; (ii) if the data in it is rarely updated and its size is large, and the query frequency is high, the prefix sum approach will be adopted for the range sum queries to it; (iii) if the data in it is updated frequently and its size is large, and the query frequency is much higher than the update frequency, the relative prefix sum approach will be employed for the range sum queries to it; (iv) if the data in it is updated frequently and its size is large, and both query and update frequencies are equal likely high, the double relative prefix sum approach will be used for range sum queries to it.

5. Conclusions

Efficient computation of range sum queries has become more important in recent years due to an increasing demand for OLAP and data cube applications. Many commercial applications

require that data cubes be updated with current information on a regular basis. For large data cubes that are updated weekly or daily, both the range query time and update time are vital to the entire system performance. In this paper, we have presented a new technique called the double relative prefix sum approach which reduces the update time complexity to $O(n^{d/3})$. Our algorithm improves the update time complexity by a factor of $O(n^{d/6})$ compared with an existing algorithm for the problem. Also, under the product model and the new cost model, the total cost for both range queries and updates by our algorithm is smaller compared to other known algorithms. Thus, our algorithm reduces the overall time complexity of the range sum query problem.

Acknowledgements

We appreciate the anonymous referees for their invaluable suggestions and comments which helped us improve the paper's quality and presentation. The work by Weifa Liang was partially supported by Australian Research Council under a small grant schema (Grant No: F00025).

References

[1] S. Agawal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, S. Sarawagi, On the computation of multidimensional aggregates, in: Proceedings of the 22nd VLDB Conference, Mumbai, India, 1996, pp. 506–521.
[2] C.-Y. Chan, Y.E. Ioannidis, Hierarchical cubes for range-sum queries, in: Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, September 1999, pp. 675–686.
[3] E.F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: an IT mandate, Technical Report, E.F Codd and Associates, 1993.
[4] P.M. Deshpande, S. Agarwal, J.F. Naughton, R. Ramakrishnan, Computation of multidimensional aggregates. Technical Report No: 1314, Department of CS, University of Wisconsin-Madison, 1996.
[5] S. Geffner, D. Agrawal, A. El Abbadi, T. Smith, Relative prefix sums: an efficient approach for querying dynamic OLAP Data Cubes, in: Proceedings of the Intenational Conference on Data Engineering, Sydney, Australia, 1999, pp. 328–335.
[6] J. Gray, A. Bosworth, A. Layman, H. Prahesh, Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals, in: Proceedings of the 12th International Conference on Data Engineering, 1996, pp. 131–139.
[7] H. Gupta, V. Harinarayan, A. Rajaraman, J.D. Ullman, Index selection for OLAP, in: Proceedings of the International Conference on Data Engineering, Birmingham, UK, 1997, pp. 208–219.
[8] V. Harinarayan, A. Rajaraman, J.D. Ullman, Implementing data cubes efficiently, in: Proceedings of the 1996 ACM SIGMOD Conference, 1996, pp. 205–216.
[9] C. Ho, R. Agrawal, N. Megiddo, R. Srikant, Range queries in OLAP Data Cubes, in: Proceedings of the 1997 ACM SIGMOD Conference, 1997, pp. 73–88.
[10] C. Ho, J. Bruck, R. Agrawal, Partial-sum queries in OLAP Data Cubes using covering codes, IEEE Trans. Comput. 47 (12) (1998) 1326–1340.
[11] T. Johnson, D. Shasha, Hierarchically split cube forests for decision support: description and tuned design, Working paper, 1996.
[12] I.S. Mumick, D. Quass, B.S. Mumick, Maintenance of data cubes and summary tables in a warehouse, in: Proceedings of the 1997 ACM SIGMOD Conference, 1997, pp. 100–111.
[13] K.A. Ross, D. Srivastava, Fast computation of sparse datacubes, in: Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997.
[14] B. Salzberg, A. Reuter, Indexing for aggregation, Working paper, 1996.
[15] S. Sarawagi, R. Agrawal, A. Gupta, On the computing the data cube. Research Report, IBM Almaden Research Center, SanJose, CA, 1996.
[16] A. Shukla, P.M. Deshpande, J.F. Naughton, K. Ramasamy, Storage estimation for multidimensional aggregates in the presence of hierarchies, in: Proceedings of the 22nd International Conference on VLDB, 1996, pp. 522–531.

[17] J.S. Vitter, M. Wang, Approximate computation of multidimensional aggregates of sparse data using wavelets, in: Proceedings of the 1999 ACM SIGMOD Conference, 1999, pp. 193–204.
[18] Y. Zhao, P.M. Deshpande, J.F. Naughton, An array-based algorithm for simultaneous multidimensional aggregates, in: Proceedings of the 1997 ACM SIGMOD Conference, 1997, pp. 159–170.

Weifa Liang received his Ph.D. degree in computer science from The Australian National University in 1998. He received his M.E. degree in computer science from University of Science and Technology of China in 1989 and his B.S. degree in computer science from Wuhan University, China in 1984. He is currently holding a lecturing position in the Department of Computer Science at The Australian National University. His research interests include parallel processing, parallel and distributed algorithms, data warehousing and OLAP, query optimazation, routing protocol design and graph theory.

Hui Wang is currently a Master (by research) student in computer science at Department of Computer Science and Electrical Engineering in The University of Queensland. She received her B.S. degree in mathematics from Anhui University, China in 1984. Before coming to Australia, as a software engineer, she has been worked in an institution in China for a decade to conduct research and development of application software in the simulation of VLSI circuits. Her current research interests include design and analysis of data warehousing, the consistency control of views in data warehousing and relational database application.

Maria E. Orlowska is currently the Professor in Information Systems at The University of Queensland in Australia. Since 1992 she has also acted as Distributed Databases Unit Leader in the Cooperative Research Centre for Distributed Systems Technology (DSTC). She was graduated with a Ph.D. (Computer Science) in June 1980 from the Institute of Applied Mathematics, Technical University of Warsaw. She is a trustee of the VLDB Endowment, and is a regular contributor to many other international conferences. She has published over 130 papers in international journals and conference proceedings. Her research expertise lies in the areas of: The Theory of Relational Databases. Distributed Databases, Various Aspects of Information Systems Design Methodologies (including Distributed Systems), Enhancement of Semantic Data Modelling Techniques by Rigorous Factors, Transaction Processing in Distributed Systems, Concurrency Control, Distributed and Federated Database Systems, and Workflows Technology.