Journal of Information Technology and Applications
Vol. 1 No. 3 December, 2006, pp. 145-154

A Recursive Relative Prefix Sum Approach to Range Sum Queries in Data Warehouses

Ye-In Chang Hue-Ling Chen Jiun-Rung Chen Fa-Jung Wu
National Sun Yat-Sen University/Dept. of Computer Science and Engineering
{changyi, chenjr, wufj}@cse.nsysu.edu.tw; chen.hueling@gmail.com

Abstract

Data warehouses contain data consolidated from several operational databases and provide the historical, and summarized data. On-Line Analytical Processing (OLAP) is designed to provide aggregate information to analyze the contents of data warehouses. An increasingly popular data model for OLAP applications is the multidimensional database, also known as data cube. A range sum query applies a sum aggregation operation over all selected cells of an OLAP data cube where the selection is specified by providing ranges of values for numeric dimensions. It is very useful in finding trends and in discovering relationships between attributes in the database. For today's applications, interactive data analysis applications which provide the current information will require fast response time and have reasonable update time. Since the size of a data cube is exponential in the number of its dimensions, it costs a lot of time to rebuild the entire data cube. To solve these updating problem, we present the *recursive relative prefix sum* method, which provides a compromise between query and update cost. From our performance study, we show that the update cost of our method is always less than that of the prefix sum method. Our recursive relative prefix sum method has a reasonable response time for ad hoc range queries on the data cube, while at the same time, greatly reduces the update cost.

Keyword: data warehouse, OLAP, range query, range sum query, update.

1. Introduction

The data warehouse is usually a read-only database for data analysis and querying processing [6, 7]. On-Line Analytical Processing (OLAP) [5] provides advanced analysis tools to extract information from data stored in a data warehouse. An increasingly popular data model for OLAP applications is the multidimensional database (MDDB), also known as data cube. In OLAP, a *range query* [3] applies a given aggregation operation over selected cells where the selection is specified as contiguous ranges in the domains of some of the attributes [4]. Such range queries are very useful in finding trends and in discovering relationships between attributes in the database. A range sum query is one aggregation operation that sums the measure attribute within the range of the query. For instance, consider a range sum query to the insurance data cube: find the total sales for customers with an age from 30 to 55, in year 1995-2001, in all of U.S., and with auto insurance. To answer this query, one way is to scan all involved cells in the data cube, and then sum them up as the results. However, since the number of cells in a data cube is exponential with its dimensions, it costs a lot of time which is proportional to the number of involved cells. Therefore, it is imperative to have a system with fast response time [8] in an interactive exploration of data cube.

Ho *et al.* [3] initialized the problem of range sum query and have presented the *prefix sum* approach to precompute prefix sums of cells in the data cube, which can then be used to answer ad hoc queries in constant time. However, when there is an update to a cell in the data cube, it requires rebuilding an array of the same size as the entire data cube. Due to the cascading update effect, the update cost is very expensive and is proportional to the entire array size. In the worst case, the update cost is $O(n^d)$ and can become impractical, where n is the size of each dimension in a d-dimensional data cube. Then, Geffner [1] proposed the *relative prefix sum* method to answer rang sum queries on data cubes. It takes $O(n^{d/2})$ time for an update and constant time for a query. However, it still incurs substantial update costs in the worst case, *i.e.*, with order of the square root of the size of the cube. In order to speed up range sum queries in data cube, Liang *et al.* [4] proposed the *double relative prefix sum* method. It requires $O(n^{1/i})$ time for each range sum query and $O(n^{d/3})$ time for each update. Geffner [2] proposed *Dynamic Data Cube*, a method that provides sublinear performance for both range sum queries and updates on the data cube. It has performance complexity of $O(\log n)$ for both queries and updates.

The above strategies have tried to improve the update problems. However, they either take more query time or need high storage cost. Thus, we propose a method, called the recursive relative prefix sum method with k-levels, which uses k relative overlay arrays and a relative prefix array. By using these components, we can provide a compromise between the range sum query cost and the update cost.

The rest of the paper is organized as follows. In Section 2, we describe the range sum query. In Section 3, we present our recursive relative prefix sum method to answer the range sum query. In Section 4, we study the performance of the recursive relative prefix sum method. Finally, we give the conclusion.

2. The Range Sum Query

Let $D = \{1,2, ...,d\}$ denote the set of dimension, where each dimension corresponds to a functional attribute [3]. A *d*-dimensional data cube can be represented by a *d*-dimensional array *A* of size $n_1 * n_2 * \ldots * n_d$, where $n_j > 2$, $j \in D$. We assume that an array has a starting index 0. For convenience, we will call each element 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 dimensions. Without loss of generality, we assume that each dimension has the same size. This allows us to present many of the formulas more concisely [1]. Thus, let the size of each dimension be *n*, and the total size of array *A* is $N = n^d$.

A range-sum query in a d-dimensional data

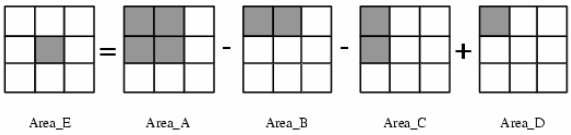


Figure 1: A geometric illustration for the two-dimensional case: Sum(Area_E) = Sum(Area_A) - Sum(Area_B) - Sum(Area_C) + Sum(Area_D)

cube is to find the sum of the values in cells that fall within the specified range. For example, in the two-dimensional case, given a measure attribute SALES and the dimensions CUSTOMER.AGE and DATE_OF_SALE, the cell at A [37, 25] contains the total sales to age 37 years old customers on day 25. A range-sum query asking for the total sales to 37 years old customers from days 20 to 22 would be answered by summing the cells A[37, 20], A[37, 21], and A[37, 22]. In general, any range-sum of *A* can be answered by accessing and combining 2^d appropriate prefix-sum. Figure 1 gives a geometrical explanation for the two-dimensional case [1]. The sum corresponding to a range query's region can be determined by adding and subtracting the sums of various other regions until we have extracted the interesting region.

3. The Recursive Relative Prefix Sum Approach

Our method makes use of *k relative overlay arrays*, which are *level-1, level-2, .. . , level-k relative overlay arrays*, respectively, and a *relative prefix (RP) array.* All of these arrays are d-dimensional arrays which are defined below.

3.1 Level-*k* Relative Overlay Arrays and Relative Prefix Array

A level-*k* relative overlay array has a set of disjoint hyper-rectangles (hereafter called "boxes") of equal size that cells of array *A* are completely partitioned into non-overlapping regions. Assume that the length of the level- *k* relative overlay box in each dimension is r_k. The size of array *A* is n^d, thus the total number of level-fc relative overlay boxes is $\lceil \frac{n}{r_k} \rceil^d$. Without loss of generality, we assume that *n* is divisible by r_k. Then, the boxes in level-*k* relative overlay array are partitioned into disjoint level-(*k* - 1) boxes. Each dimension of a level-(*k* - 1) box has the same size $r_{k-1}r_k$. Thus, there are $\lceil \frac{n}{r_{k-1}r_k} \rceil^d$ level-(*k* - 1) relative overlay boxes. Each level-(*k* - 1) relative overlay box covers r_{k-1}^d level-*k* relative overlay boxes. Following this recursive partition rule, we can find out all these *k*

Array A								
Index	0	1	2	3	4	5	6	7
0	4	1	1	1	2	3	3	4
1	2	9	3	2	3	2	6	8
2	5	3	2	6	3	6	1	7
3	3	8	2	3	1	3	2	6
4	4	7	1	5	2	2	2	5
5	6	6	1	7	2	7	1	4
6	8	3	1	9	3	8	1	3
7	3	4	6	1	4	3	3	2

Figure 2: A two-dimensional data cube represented as a two-dimensional array *A*

Index	0	1	2	3	4	5	6	7
0	0	0	5	0	0	0	5	0
1	0		11		0		5	
2	6	10	28	3	5	5	28	12
3	0		11		0		4	
4	0	0	11	0	0	0	4	0
5	0		12		0		9	
6	10	13	36	12	4	9	27	9
7	0		7		0		7	

Figure 3: The level-2 relative overlay array of the data cube *A*

different level overlay arrays.

A level-*j* relative overlay box is *anchored* at $(a_1, a_2, ..., a_d)$ if the box corresponds to the region of array *A* where $(a_1, a_2,..., a_d)$ is the lowest index of a cell at each dimension. We denote the level-j relative overlay box as $B_j[a_1, a_2, \ldots, a_k]$, for $j \in \{1,2,... ,k\}$. A

level-*j* relative overlay box $B_j[a_1, a_2, \ldots, a_d]$ is said to cover a cell $(x_1, x_2, \ldots, x_d)$ in array *A*, if the cell falls within the boundaries of the level-j relative overlay box, i.e., if $\forall i((a_i \leq x_i) \bigwedge (a_i + r_k \geq x_i))$. Each level-*j* relative overlay box corresponds to an area of array *A* of size $(r_j * r_{j+1} * \ldots * r_k)^d$ cells, for $j \in \{1, 2, \ldots, k\}$.

We take an 8 x 8 array *A* in Figure 2 as an example to illustrate a recursive relative prefix sum method with 2-levels as follows. Each level-2 relative overlay box in Figure 3 is bounded by bold lines is of size 2x2, *i.e.*, $r_2 = 2$. The total number of level-2 relative overlay boxes is $\lceil \frac{n}{r_k} \rceil^d = (8/2)^2 = 16$. The boxes are anchored at cells (0,0), (0,2), (0,4), (0,6), (2,0), (2,2), (2,4), (2,6), (4,0), (4,2), (4,4), (4,6), (6,0), (6,2), (6, 4), and (6, 6). Each level-2 relative overlay box covers $2^2 = 4$ cells of array *A*. In Figure 4, each level-1 overlay box which is bounded by bold lines is of size 4x4, i.e., $r_1 r_2 = 4$ and $r_1 = 2$. The total number of level-1 relative overlay boxes is $\lceil \frac{n}{r_{k-1} r_k} \rceil^d = (8/4)^2 = 4$. The boxes are anchored at cells (0,0), (0,4), (4,0), and (4,4). Each level-1 relative overlay box covers $(r_1 r_2)^2 = 16$ cells of array *A* and $r_1^2 = 4$ level-2 relative overlay boxes.

Index	0	1	2	3	4	5	6	7
0	4	5	1	2	2	5	3	7
1	6	16	4	7	5	10	9	21
2	5	8	2	8	3	9	1	8
3	8	19	4	13	4	13	3	16
4	4	11	1	6	2	4	2	7
5	10	23	2	14	4	13	3	12
6	8	11	1	10	3	11	1	4
7	11	18	7	17	7	18	4	9

data cube *A*

Each level-j relative overlay box stores an *anchor value* V_j, plus $(r_j^d - (r_j - 1)^d) - 1$ *border values*. The other cells covered by the level-j relative overlay box are not needed in the level-j overlay, and would not be stored. Values stored in a level-*j* relative overlay box provide sums of regions outside the box, but within its level-*(j* - 1) relative overlay box, for $j \in \{1, 2, \ldots, k\}$. Note theat $j = 1$, level-0 means the region of entire *A*. The anchor value of a level-j relative overlay box is the sum of all cells in *A* up, but not including, the cell under V_j within its level-*(j* - 1) relative overlay box.

For example, there are 16 level-2 relative overlay boxes of size 2 x 2 in Figure 3. The anchor value at cell (6, 6) and the border values at cells (6, 7) and (7, 6) are stored in one of the level-2 relative overlay boxes. In Figure 4, there are 4 level-1 relative overlay boxes of size 4x4. The anchor value at cell (4, 4) and the border values at cells (4, 5), (4, 6), (4, 7), (5,4), (6,4), and (7, 4) are stored in one of the level-1 relative overlay boxes.

The relative prefix array *(RP)* has the same size as array *A*. It is partitioned into regions of cells that correspond to level-*k* relative overlay boxes. A cell $RP[x_1,x_2, \ldots,x_d] = Sum(A[b_1,b_2, \ldots, b_d] : A[x_1, x_2, \ldots, x_d])$ is covered by a level-2 relative overlay box B_2, where $(b_1, b_2, \ldots, b_d)$ is the lowest index of a cell at each dimension. Each region in *RP* contains prefix sums that are relative to the area enclosed by the level-fc relative overlay box. Each region of *RP* is independent of other regions.

Figure 5 shows a relative prefix array *RP*, where $r_2 = 2$ and the shaded area denotes the difference between the proposed approach and the relative prefix sum method. Take Figure 2 as an example. $RP[4,7] = Sum(A[4, 6] : A[4, 7]) = A[4, 6] + A[4, 7] = 3 + 4 = 7$, $RP[5, 6] = Sum(A[4, 6] : A[5, 6]) = A[4, 6] + A[5, 6] = 3 + 8 = 11$ and $RP[5, 7] = Sum(A[4, 6] : A[5, 7]) = A[4, 6] + A[4, 7] + A[5, 6] + A[5, 7] = 3+4+8+3 = 18$. The lowest index of a cell at each dimension in the level-2 relative overlay box which contains these cells is cell (4, 6).

Now we give a formal definition of anchor value and border value as follows. Let $(b_1, b_2, \ldots, b_d)$ be the lowest index of a cell at each dimension in a level-2 relative overlay box B_2, and $(a_1, a_2, \ldots, a_d)$ be the lowest index of a cell at each dimension in a level-1 relative overlay box *B1*. The variable V_2 is denoted as the level-2 anchor value, where $V_2[b_1, b_2, \ldots, b_d] = Sum(A[a_1, a_2, \ldots, a_d] : A[b_1,b_2, \ldots, b_d] - A[b_1, b_2, \ldots, b_d]$, and B_2 is covered by B_1. Take Figure 3 as an example. The cell [4, 6] is one of the lowest index of a cell at each dimension in the level-2 relative overlay box. Then, $V_2[4, 6] = Sum(A[4, 4] : A[4, 6]) - A[4, 6] = A[4, 4]+,4[4, 5]+,4[4, 6]-A[4, 6] = 2+2+3-3 = 4$. The level-2 border value contained in cell $(x_1, x_2, \ldots, x_d)$ is equal to

$$Sum(A[a_1, a_2, \ldots, a_d] : A[x_1, x_2, \ldots, x_d]) - RP[x_1, x_2, \ldots, x_d] - V_2,$$

where V_2 is the anchor value of the level-2 relative overlay box, $RP[x_1, x_2, \ldots, x_d]$ is the value of this cell in array *RP*, and $(a_1, a_2, \ldots, a_d)$ is the lowest index of a

Index	0	1	2	3	4	5	6	7
0	0	0	0	0	7	0	0	0
1	0				16			
2	0				32			
3	0				48			
4	14	21	29	41	81	14	26	51
5	0				20			
6	0				41			
7	0				55			

Figure 5: The relative prefix array *RP* of the data cube *A*

cell at each dimension in a level-1 relative overlay box which covers this level-2 relative overlay box. Take Figure 3 as example. The level-2 border value contained in cell (5, 6) is equal to *Sum(A*[4,4] : *A*[5,6]) - RP[5, 6] - V_2 = *Sum(A*[4,4] : *A*[5,6]) - *(Sum(A*[4,6] : *A*[5,6]) - *((Sum(A*[4, 4] : *A*[4, 6]) - *A*[4, 6]) = *Sum(A*[5, 4] : *A*[5, 5]) = 9. The level-2 border value at the row dimension can be computed similarly.

The level-1 relative overlay box anchored at (a_i, a_2, .. ., a_d) has a level-1 anchor value Vi[a_1, a_2,. .., a_d] = *Sum(A*[0, 0,. .., 0] : *A*[a_1,a_2,. .. ,a_d]) - *A*[a_1,a_2,. .. ,a_d]. And the level-1 border value contained in cell (y_1, y_2, …, y_d) is equal to

$$Sum(A[0,0,\ldots,0] : A[y_1,y_2,\ldots,y_d]) - Sum(A[a_1,a_2,\ldots,a_d] : A[y_1,y_2,\ldots,y_d]) - V_1,$$

where V_1 is the anchor value of the level-1 relative overlay box. Figure 6 shows a level-1 relative overlay box superimposed on array *A*. Similarly, the level-1 anchor value of the level-1 relative overlay box is equal to the sum of the shaded region in array *A*. Figure 7 shows the calculation of level-1 border values. The level-1 border values are equal to the sum of the associated shaded regions of array *A*.

Take Figure 4 as an example. The cell [4,4] is one of the lowest index of a cell at each dimension in

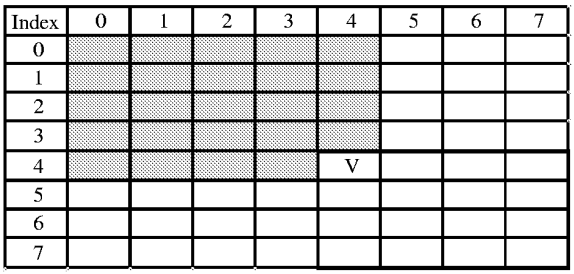


Figure 6: Array *A* showing calculation of a level-1 relative overlay box anchor value

a level-1 relative overlay box. Then, V_i[4,4] = *Sum(A*[0,0] : A[4,4]) - A[4,4] = 81. The level-1 border value Y_1 contained in cell (4, 5) is equal to *Sum(A*[0, 0] : *A*[4, 5]) - *Sum(A*[4, 4] : *A*[4, 5]) - V_1 = 20. The level-1 border value Y_2 contained in cell (4,6) is equal to *Sum(A*[0,0] : *A*[4, 6]) - *Sum(A*[4, 4] : *A*[4, 6]) - V_1 = Y_1 + *Sum(A*[0,6] : *A*[3,6]*)* = 41. The level-1 border value Y_3 contained in cell (4,7) is equal to *Sum(A*[0,0] : *A*[A,7]) - *Sum*(*A*[4,4] : *A*[4,7]) - V_1 =Y_2 + *Sum(A*[0, 7] : *A*[3, 7]) = 55. The level-1 border values X_1, X_2, and X_3 at the column dimension can be computed similarly.

3.2 Range Sum Queries and Updates

According to Figure 1, Figure 8 shows the construction of a complete region sum using the recursive relative prefix sum method with 2-levels. In Figure 8, * denotes an arbitrary cell in array *A* at cell (3, 7), as shown in 2. For reference, the level-1 relative overlay box and level-2 relative overlay box covering this cell have been superimposed on array *A*. The level-1 anchor and level-1 border values from the level-1 relative overlay box provide the sum of the portion of the light-dark region outside the level-1 relative overlay box. The level-2 anchor and level-2 border values from the level-2 relative overlay box provide the sum of the portion of the diagonal region between the level-1 relative overlay box and the level-2 relative overlay box. The cell * in *RP* provides the sum of the portion of the deep-dark region within the level-2 relative overlay box. Sum of these three shaded regions together yields the sum of all cells in array *A* that fall within the shaded region. Following this manner, any region sum rooted at *A*[0,0,..., 0] could be generated, and it is sufficient to provide the region sums required by the method illustrated in Figure 1.

Take Figure 2 as an example. Suppose a range sum query is given which ranges from cell .A[0,0] to cell *A*[3,7] in array *A*. We find that cell [3, 7] is covered by the level-1 relative overlay box anchored at cell [0,4] in Figure 4 and level-2 relative overlay box anchored at cell [2, 6] in 3. First, we calculate the sum of the area that falls outside the level-1 relative

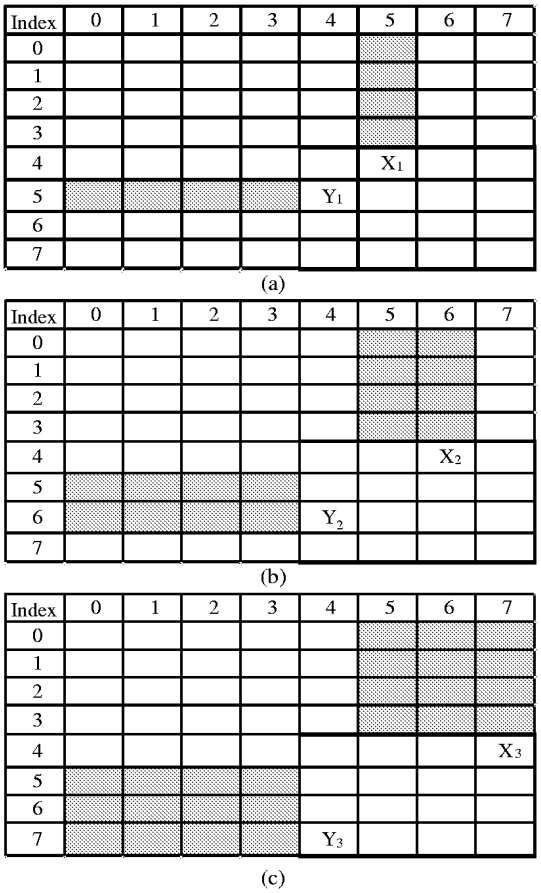
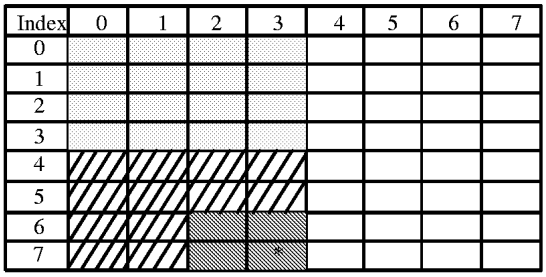


Figure 7: Array *A* showing calculation of level-1 relative overlay box border values: (a) level-1 border values X_1 and Y_1; (b) level-1 border values X_2 and Y_2; (c) level-1 border values X_3 and Y_3.



level-1 relative overlay, level-2 relative overlay and *RP*

overlay box, which is the sum of a level-1 anchor value and two level-1 border values. The level-1 anchor value of the level-1 relative overlay box which is anchored at cell (0, 4) is 14. Since cell (3, 7) is three cells to the right of, and three cells down from the level-1 anchor cell, there are two level-1 border values X_3 and Y_3 which are 41 and 0, respectively. Therefore, the area outside the level-1 relative overlay box has a value equal to 14+41+0. Then, we calculate the sum of the area that falls between the level-1 relative overlay box and the level-2 relative overlay box, which is the sum of a level-2 anchor value and two level-2 border values. The level-2 anchor value of the level-2 relative overlay box which is anchored at cell (2, 6) is 36. Since cell (3, 7) is one cell to the right of, and one cell down from the level-2 anchor cell, there are two level-2 border values which are 12 and 7, respectively. Therefore, the area between the level-1 relative overlay box and the level-2 relative overlay box has a value equal to 36 + 12 + 7. Finally, we calculate the sum of the cells inside the level-2 relative box which is anchored at cell (2,6). This sum is stored in *RP*[3, 7] as shown in Figure 5, and its value is 17. Thus, the complete range sum for the region *A*[0, 0] : A[3, 7] is 14 + 41 + 0 + 36 + 12 + 7 + 17 = 127.

3.2.1 Updates

Assume that a cell in the data cube has been updated. Updating this cell requires updates to the data structures. Updates in *RP* are constrained to a region covered by one level-2 relative overlay box. In other words, every cell in the level-2 relative overlay box whose index is larger than the index of the updated cell will be updated. Then, we update the content of the level-2 relative overlay boxes covered by a level-1 relative overlay box in which the updated cell is contained. That is, we update the level-2 border values and level-2 anchor values of some level-2 relative overlay boxes within the level-1 relative overlay box. For example, as shown in Figure 9, * denotes the location of the changed cell. The shaded region to the right of the changed cell shows the level-2 relative overlay box whose border values in the first dimension will be affected by the update. In this example, only level-2 relative overlay cell (2,1) will be updated. The shaded region below the changed cell shows the level-2 relative overlay box whose border values in the second dimension will be affected by the update. In this example, level-2 relative overlay cell (1, 2) will be updated. Every level-2 anchor value whose index is larger than the changed cell will be affected by the update, too. In this case, level-2 relative overlay cell (2, 2) will be affected.

Finally, the content of every level-1 relative overlay box whose cells' lowest index is larger than the lowest index of cells within the level-1 relative overlay box covering the changed cell must be updated; i.e., we update the level-1 border values and level-1 anchor values of some level-1 relative overlay boxes. For example, as shown in Figure 10, * denotes the location of the changed cell. The shaded region to the right of the changed cell shows the level-1 relative overlay box whose border values in the first dimension will be affected by the update. In this example, level-1 relative overlay cells (4,1), (4, 2),

Index	0	1	2	3	4	5	6	7
0	0	0	5	0	0	0	5	0
1	0	*	11		0		5	
2	6	10	28	3	5	0	28	12
3	0		11		0		4	
4	0	0	11	0	0	0	4	0
5	0		12		0		9	
6	10	13	36	12	4	9	27	9
7	0		7		0		7	

Figure 9: An update example of level-2 relative overlay array

Index	0	1	2	3	4	5	6	7
0	0	0	0	0	7	0	5	0
1	0	*			16			
2	0				32			
3	0				48			
4	14	21	29	41	81	14	26	51
5	0				20			
6	0				41			
7	0				55			

Figure 10: An update example of level-1 relative overlay array

and (4,3) will be updated. The shaded region below the changed cell shows the level-1 relative overlay box whose border values in the second dimension will be affected by the update. In this example, level-1 relative overlay cell (1,4), (2,4), and (3,4) will be updated. Every level-1 anchor value whose index is larger than the changed cell will be affected by the update, too. In this case, level-1 anchor cell (4, 4) will be affected.

Algorithms for query and update operations in the recursive relative prefix sum method with more

than 2-levels are similar to those in the recursive relative prefix method with 2-levels. So we do not describe the algorithms further. By recursively creating boundaries with appropriate *k* levels, we can get the sum of each region in constant time.

4. Performance Study

In this section, we study the performance of our proposed recursive relative prefix sum method for the range sum query problem, and make a comparison with other range sum query strategies, including the *prefix sum method* [3], the *relative prefix sum method* [1], the *double relative prefix sum method* [4], and the *dynamic data cube method* (DDC) [2].

4.1 Performance Analysis

In the recursive relative prefix sum method with 2-levels, choosing different values of r_1 and r_2 can make quite different update costs, where r_2 is the length of the level-2 relative overlay box in each dimension and r_1 means a level-1 relative overlay box covering r_1 level-2 relative overlay boxes in each dimension. When the values of r_1 and r_2 are smaller, fewer cells in RP will be updated, but more cells in the level-2 and level-1 relative overlay arrays will be updated, which would result in a larger total number of affected cells. When the values of r_1 and r_2 are larger, fewer cells in the level-2 and level-1 relative overlay arrays will be updated; however, more cells in RP will be updated. This may lead to a larger total number of affected cells. We make different combination of values of r_1 and r_2. Then, we observe that update costs would be smallest when we choose $r_1 = r_2 = 16$.

For the the recursive relative prefix sum method with 2-levels, each level-2 relative overlay box is of size r_2 and each level-1 relative overlay box covers $(r_1 r_2)^d$ cells of array *A* and r_1^d level-2 relative overlay boxes. Let $U = (u_1, u_2, \ldots, u_d)$ be the updated cell, *B2* be the level-2 relative overlay box anchored at $(b_1, b_2, \ldots, b_d)$ that covers *U* and B_1 be the level-1 relative overlay box anchored at $(a_1, a_2, \ldots, a_d)$ that covers #2. The cost of updating *U* requires updates to the overlay cells and *RP*. Updates in *RP* are constrained to a region covered by level-2 relative overlay box #2. In the worst case, the cost of updating *RP* is r_2^d. Next, we describe which level-2 relative overlay cells and level-1 relative overlay cells need to be updated. Generally speaking, these are all level-2 relative overlay cells and level-1 relative overlay cells that include *U* in their aggregation. Let M_i denote the coordinates of the anchors of the affected level-1 relative overlay boxes in the i-dimension. Thus, we have $M_i = \{a_i + r_1 r_2;\ a_i + 2r_1 r_2;\ a_i + 3r_1 r_2,\ \ldots,\ n - r_1 r_2\}$. Let N_i denote the coordinates of the affected level-1 relative overlay cells inside a certain level-1 relative overlay box in the i-dimension. Then, we have $N_i = \{u_i,\ u_i + 1;\ u_i + 2;\ \ldots,\ a_i + (r_1 r_2 - 1)\}$. Let $P = (p_1, p_2, \ldots, p_d)$ denote the level-1 relative overlay cells that need to be updated. These level-1 relative overlay cells satisfy

$$\forall i : p_i \in \begin{cases} M_i & , \text{if } u_i = a_i \\ M_i \bigcup N_i & , \text{otherwise} \end{cases}$$

and do not belong to level-1 relative overlay box B_1. When we choose $u_i \neq a_i$ and u_i as small as possible (the smaller u_i, the smaller dj, i.e., the more elements in M_i), the largest number of affected level-1 relative overlay cells would be obtained. Therefore, in the worst case, M_i has at most $n / r_1 r_2 - 1$ elements, and *Ni* has up to $r_1 r_2 - 1$ elements. Consequently, there are at most $(n / r_1 r_2 - 1 + r_1 r_2 - 1) = (n / r_1 r_2 + r_1 r_2 - 2)$ possible values for each dimension, and there are at most $(n / r_1 r_2 + r_1 r_2 - 2)^d$ level-1 relative overlay cells that satisfy the above formula. Within these cells, $(r_1 r_2 - l)^d$ cells fall into level-1 relative overlay box B\. In the worst case, $(n / r_1 r_2 + r_1 r_2 - 2)^d - (r_1 r_2 - 1)^d$ level-1 relative overlay cells must be updated.

Index	0	1	2	3	4	5	6	7
0	0	0	0	0	7	0	5	0
1	0	*			16			
2	0				32			
3	0				48			
4	14	21	29	41	81	14	26	51
5	0				20			
6	0				41			
7	0				55			

Figure 11: The level-1 relative overlay array of the data cube *A*

For example, as shown in Figure 11, we assume that cell (1,1)(= *U)* is updated. Then, we have $M_x = \{4\}$, $N_x = \{1;2;3\}$, $M_y = \{4\}$, and $N_y = \{1;2;3\}$. The affected level-1 relative overlay cells are all cells (p_x, p_y) where $p_x \in \{1;2;3;4\}$ and $p_y \in \{1;2;3;4\}$, minus (1,1), (1, 2), (1, 3), (2,1), (2, 2), (2, 3), (3,1), (3, 2) and (3, 3) which fall into the same level-1 relative overlay box as (1,1). Thus, in this worst case, $(n / r_1 r_2 + r_1 r_2 - 2)^d - (r_1 r_2 - l)^d = (8/4 + 4 - 2)^2 - (4 - 1)^2 = 7$ level-1 relative overlay cells must be updated.

Next, we discuss which level-2 relative overlay cells need to be updated. Let S_i denote the coordinates of the anchors of the affected level-2 relative overlay boxes in the i-dimension. Thus, we have $S_i = \{b_i + r_2;\ bi + 2r_2;\ bi + 3r_2, \ldots, b_i + r_1 r_2 - r_2\}$. Let T_i denote the coordinates of the affected level-2 relative overlay cells inside a certain level-2 relative overlay box in the i-dimension. Then, we have $T_i = \{u_i, u_i + 1;\ u_i + 2;\ \ldots,\ b_i + (r_2 - 1)\}$. Let $Q = (q_1, q_2, .., q_d)$ denote the level-2 relative overlay cells that need to be updated. These level-2 relative overlay cells satisfy

$$\forall i : q_i \in \begin{cases} S_i & , \text{if } u_i = b_i \\ S_i \bigcup T_i & , \text{otherwise} \end{cases}$$

and do not belong to level-2 relative overlay box B_2. Similarly, when we choose $u_i \neq b_i$ and *Ui* as small as possible, the largest number of affected level-2 relative overlay cells would be obtained. Therefore, in the worst case, S_i has at most $r_1 - 1$ elements, and T_i has up to r_2 - 1 elements. Consequently, there are at most (r_1 - 1 + r_2 - 1) = (r_1 + r_2 - 2) possible values for each dimension, and there are at most (r_1 + r_2 - 2)^d level-2 relative overlay cells that satisfy the above formula. Within these cells, (r_2 - 1)^d cells fall into level-2 relative overlay box B_2. In the worst case, (r_1 + r_2 - 2)^d - (r_2 - *1*)^d level-2 relative overlay cells must be updated.

For example, as shown in Figure 12, we assume that cell (1,1)(= *U)* is updated. Then, we have S_x = {2}, N_x = {1}, M_y = {2}, *and* N_y = {1}. The affected level-2 relative overlay cells are all cells *(q_x , q_y)* where $q_x \in$ {1; 2} and $q_y \in$ {1; 2}, minus (1,1) which fall into the level-2 relative overlay box. In this worst

Index	0	1	2	3	4	5	6	7
0	0	0	5	0	0	0	5	0
1	0	*	11		0		5	
2	6	10	28	3	5	5	28	12
3	0		11		0		4	
4	0	0	11	0	0	0	4	0
5	0		12		0		9	
6	10	13	36	12	4	9	27	9
7	0		7		0		7	

Figure 12: The level-2 relative overlay array of the data cube *A*

case, (r_1 +r_2 - 2)^d - (r_2 - 1)^d = (2 + 2 - 2)^2 - (2 - l)^2 = 3 level-2 relative overlay cells must be updated.

Therefore, the worst update cost is obtained, when cell (1,1,..., 1) is updated, resulting in a cost of

$$
\begin{aligned}
& (\tfrac{n}{r_1 r_2} + r_1 r_2 - 2)^d - (r_1 r_2 - 1)^d \\
& \text{(level-1 relative overlay array)} \\
+\ & (r_1 + r_2 - 2)^d - (r_2 - 1)^d \\
& \text{(level-2 relative overlay array)} \\
+\ & (r_2 - 1)^d \\
& \text{(RP)} \\
=\ & (\tfrac{n}{r_1 r_2} + r_1 r_2 - 2)^d - (r_1 r_2 - 1)^d + (r_1 + r_2 - 2)^d.
\end{aligned}
$$

This value is minimized for $r_1 = r_2 = \lceil n^{1/3} \rceil$.

Similarly, in the recursive method with 3-levels, the worst update cost could be obtained, when cell (1,1,..., 1) is updated, resulting in a cost of

$$
\begin{aligned}
& (\tfrac{n}{r_1 r_2 r_3} + r_1 r_2 r_3 - 2)^d - (r_1 r_2 r_3 - 1)^d \\
& \text{(level-1 relative overlay array)} \\
+\ & (r_1 + r_2 r_3 - 2)^d - (r_2 r_3 - 1)^d \\
& \text{(level-2 relative overlay array)} \\
+\ & (r_2 + r_3 - 2)^d - (r_3 - 1)^d \\
& \text{(level-3 relative overlay array)} \\
+\ & (r_3 - 1)^d \\
& (RP) \\
=\ & (\tfrac{n}{r_1 r_2 r_3} + r_1 r_2 r_3 - 2)^d - (r_1 r_2 r_3 - 1)^d + (r_1 + \\
& r_2 r_3 - 2)^d - (r_2 r_3 - 1)^d + (r_2 + r_3 - 2)^d.
\end{aligned}
$$

And this value is minimized for $r_1 = r_2 = r_3 = \lceil n^{1/4} \rceil$

4.2 Simulation Study

For a given data cube, we assume that each range query takes time t_q and each update takes time t_u. Both t_q and t_u are the time in the worst case. For a given time window, we 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, the total time used for both range queries and updates is $n_q t_q + n_u t_u$ [4]. We consider not only in the worst case but also in the general case. Let $c = n_q / n_u$ which

Table 1: Parameters used in the simulation

n	length of each dimension
r_i	length of the level-i relative overlay box of recursive relative prefix sum method in each dimension
c	ration of the times of query and update
k	recursive relative prefix sum method with k-levels

is the ratio of query and update. Thus, $n_q t_q + n_u t_u = n_u(ct_q + t_u)$, since $n_q = cn_u$. In the general case, we take the average of $ct_q + t_u$ as our average cost, and both t_q and t_u are the time on average here. We assume that the time of accessing a cell is the same regardless of query or update. Thus, t_q and t_u are equal to how many cells they access on average. The parameters used in our simulation are shown in Table 1.

If we could reduce the dependencies between border values of level-1 relative overlay boxes, the update cost can be significantly improved. Therefore, We apply the *Cumulative B Tree* (B^c tree) [2] to reduce the cascading update that occurs when an individual row sum is updated. There will be a separate B^c tree for each set of level-i border values, where i$\in${1,2, ..., k - 1}, in the recursive relative prefix sum method with fc-levels.

Figure 13 shows a B^c tree for one set of border values in a level-i relative overlay box. The B^c tree modifies the standard S-tree in two ways. The first

modification is with regard to keys. Each leaf of the B^c tree corresponds to one border value cell in a level-i relative overlay box. The key for each leaf is not equal to the data value in the cell, but rather is equal to the index of the cell in the one-dimensional array of level-i border values. Thus, the leaves of the B^c tree are in the same order as the border value cells in the level-i relative overlay box. The first leaf in the figure corresponds to the first border value cell of the level-i relative overlay box. Its key is thus 1, and it stores the value 8, which is the sum of all cells in the column above the first border value cell of the level-1 relative overlay box. The second leaf correspond to the second border value cell; its key is thus 2, and its value is (14 - 8 = 6), which is the sum of all cells in the column above the second border value cell of the level-i relative overlay box. A B^c tree also augments the standard B-tree by storing additional values in interior nodes. For each node entry, the subtree sum (STS) stores the sum of the subtree found by following the left branch associated with the entry. The fanout of the tree in the figure is three, so there are at most two STS values in each node. However, for fanout *f*, there are (*f* - 1) STSs. In this example, the root stores an STS of 23, which represents the sum of the leaf values in the left subtree below the

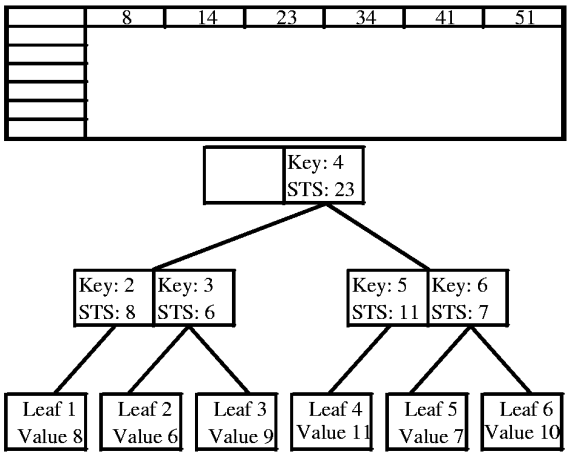


Figure 13: One set of border values stored in a B^c tree

root (8 + 6 + 9 = 23). The interior node with key 6 has an STS of 7, which represents the sum of the leaf values in its left subtree (7).

A border value is obtained from the B^c tree in O(log *r*) steps, where *r* is the number of border values in the level-*i* relative overlay box. For example, as shown in Figure 13, we wish to find the value of border value cell 5 in the level-*i* relative overlay box. We start at the root, using 5 as the key. 5 is in the right subtree of the root. The STS of 23 precedes it, so we add 23 to our total and descend to the right child of the root. 5 is in the middle subtree of this node. The node has two STSs, 11 and 7. The STS 11 precedes the subtree we will descend, so we add it to our total. The STS 7 is after the subtree we will descend, so we ignore it. We descend to the leaf, which contains the value 7, and add it to our total, producing 23 + 11 + 7 = 41. In the worst case, the query time of the B^c tree requires O(log *r*).

Similarly, as shown in Figure 13, we suppose an update to the data cube causes the border value cell 3 to change from 9 to 12. To update the B^c tree to reflect this change, we will use a bottom-up method. We begin by traversing down the tree to the leaf, where we note that the difference between the old and new value is +3. After we update the value of cell 3 with the new value, we will return up the tree and update one STS value per visited node with the difference, when appropriate. In this case, we first ascend to the node with key 3 in tree level 1. We do not update the STS value of this node because the changed cell does not fall in its left subtree. We next ascend to the root, we update the STS value in the root with the difference, producing 23 + 3 = 26. At most one STS value will be modified per visited node during the update process. Thus, updating the B^c tree requires O(log *r*).

The B^c tree breaks the barrier to efficient updates of border values in one dimension. Now, let us consider the general case, where the dimensionality of the data cube is greater than two. In general, a level-i relative overlay box of *d* dimensions has *d* groups of border values, and each group is *(d* - 1) dimensional. Thus, the level-i relative overlay box values of a *d*-dimensional data cube can be stored as *(d* - l)-dimensional data cubes, recursively; when *d* = 2, we use the B^c tree to store the border values. Algorithms for query and update are as described before, except that border values of level-*i* relative overlay are not accessed directly from arrays; rather, they are obtained from secondary trees. In the recursive relative prefix sum method with 2-levels, the worst update case will affect $d(\frac{n}{r_1 r_2})(r_1 r_2^{d-1})$ border cells of level-1 relative overlay originally, and therefore, we could reduce affected level-1 border cells to $(\frac{n}{r_1 r_2})d! \log r_1 r_2$. Thus, in the worst update case, an update to the data cube will affect $(r_2 - 1)^d$ cells in the *RP* array + $dr_1(r_2^{d-1})$ border cells of level-2 relative overlay + $(r_1 - 1)^d$ level-2 anchor cells + $(\frac{n}{r_1 r_2})d! \log r_1 r_2$ border cells of level-1 relative overlay + $(\frac{n}{r_1 r_2} - 1)^d$ level-1 anchor cells. This formula can be reasonably approximated as

$$r_2^d + dr_1 r_2^{d-1} + r_1^d + (\frac{n}{r_1 r_2})d! \log r_1 r_2 + (\frac{n}{r_1 r_2})^d.$$

By using approximation, we find that the cost is minimized when $r_1 = r_2 = \lceil n^{1/3} \rceil$, and the time for the worst update cost therefore is O($n^{d/3}$).

Since we apply the B^c tree to the border val¬ues of level-i relative overlay boxes, the query time time is not a constant time anymore. In the re¬cursive

relative prefix sum method with 2-levels, calculating each region sum requires adding one level-1 anchor value, d level-1 border values, one level-2 anchor value, d level-2 border values, and one value from RP. Thus, when $r_1 = r_2 = \lceil n^{1/3} \rceil$, the query time is O(log*n*). Table 2 presents the performance complexities of various methods of computing range sum queries.

Similarly, the parameters used in our simulation are like previous subsection. But here, we apply the B^c tree structure to the border values of level-1 relative overlay boxes in the recursive relative prefix sum method with 2-levels, and the border values of level-1, level-2 relative overlay boxes in the recursive relative prefix sum method with 3-levels. The details of the comparison for the average cost is summarized in Table 3. From Table 3, we observe that the average cost of our proposed method is better than the others. When we consider the ratio *c* of the times of query and updates, Figure 14 shows the result of the simulation. As c increases gradually, the average cost of our proposed method increases gently, and has better average cost performance than the others.

Table 2: Performance complexities of various methods

Method	Query	Update	Time for *q* queries and *p* updates
Native	$O(n^d)$	O(l)	$O(p + qn^d)$
Prefix sum	O(l)	$O(n^d)$	$O(pn^d + q)$
Relative prefix sum	O(l)	$O(n^{d/2})$	$O(pn^{d/2} + q)$
Double relative prefix sum	$O(n^{1/3})$	$O(n^{d/3})$	$O(pn^{d/3} + qn^{1/3})$
DDC	$O(\log^d n)$	$O(\log^d n)$	$O(p\log^d n + q\log^d n)$
Recursive relative prefix sum method with 2-levels	O(log *n*)	$O(n^{d/3})$	$O(pn^{d/3} + q$ log *n*)

5. Conclusion

For large data cubes in OLAP that are updated weekly or daily, effective performances for both range query time and update time are essential. In this paper, we have proposed a new method called recursive relative prefix sum method with fc-levels which creates boundaries recursively that limit cascading updates to distinguished cells. From our performance study, we have shown that the update cost of our method is always less than that of the prefix sum method. Our recursive relative prefix sum method has a reasonable response time for ad hoc range queries on the data cube and greatly reduces the update cost.

References

[1] S. Geffner, D. Agrawal, A. El Abbadi and T. Smith, 1999, "Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes," *Proc. of the 15th IEEE Int. Conf. on Data Eng.*, pp. 328-335.

[2] S. Geffner, D. Agrawal and A. El Abbadi, 2000, "The Dynamic Data Cube," *Proc. of the 7th Int. Conf. on Extending Database Technology*, pp. 237-248.

[3] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo and Ramakrishnan Srikant, 1997, "Range Queries in OLAP Data Cubes," *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pp. 73-88.

[4] W. Liang, H. Wang and M. E. Orlowska, 2000, "Range Queries in Dynamic OLAP Data Cubes," *Trans, on Data and Knowledge Eng.*, Vol. 34, No. 1, pp. 21-38.

[5] Z. Liu and Wesley W. Ch, 2005, "Knowledge-Based Query Expansion to Support Scenario-Specific Retrieval of Medical Free Text," *Proc. of ACM Symp. on Applied computing*, pp. 1076-1083.

[6] E. Malinowski and E. Zimanyi, 2006, "A Conceptual Solution for Representing Time in Data Warehouse Dimensions," *Proc. of the 3rd Asia-Pacific Conf. on Conceptual Modelling*, pp. 45-54.

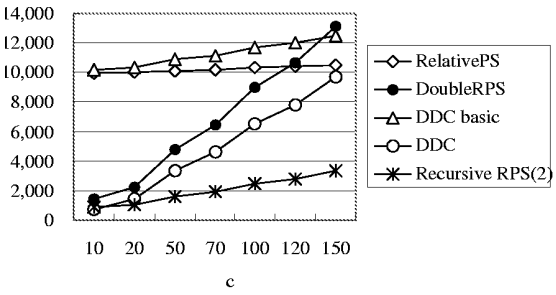
Table 3: A comparison of the average cost between different strategies

Strategy	Average query cost	Average update cost	Average cost	Storage cost
Prefix Sum	1	25,076,462	25,076,562.84	100,000,000
Relative Prefix Sum	4	9,997	10,397.59	101,990,000
Double Relative Prefix Sum	83.71	590	8,962.51	109,307,466
DDC (basic)	16.9	10,018	11,709.14	184,423,020
DDC	64.22	108	6,531.87	184,423,020 $+\omega^1$
Recursive Relative Prefix Sum with 2-levels	17.88	673	2,462.28	109,312,534 $+\alpha^2$
Recursive Relative Prefix Sum with 3-levels	29.64	231	3,196.36	121,189,900 $+\beta^3$

[1]For fanout $f = 3$, there are $60,352,728$ internal nodes. Thus, $\omega = 60,352,728 \times \nu$, where ν is the storage cost of a node.

[2]For fanout $f = 3$, there are $210,084$ internal nodes. Thus, $\alpha = 210,084 \times \nu$, where ν is the storage cost of a node.

[3]For fanout $f = 3$, there are $1,120,400$ internal nodes. Thus, $\beta = 1,120,400 \times \nu$, where ν is the storage cost of a node.



* Recursive RPS(2): Recursive Relative Prefix Sum method with 2-levels

Figure 14: A comparison of the average cost by using different *c*

[7] Mikael R. Jensen, Thomas Holmgren and Torben Bach Pederse, 2004, "Discovering Multidimensional Structure in Relational Data," *Proc. of the 6th Int. Conf. on Data Warehousing and Knowledge Discovery,* pp. 138-148.

[8] Tok Wang Ling, Wai Chong Low, Zhong Wei Luo, Sin Yeung Lee and Hua-Gang Li, 2002, "Variable Sized Partitions for Range Query Algorithms," *Proc. of the 13th Int. Conf. on Database and Expert Systems Applications,* pp. 193-202.