Relative Prefix Sums:
An Efficient Approach for Querying Dynamic OLAP Data Cubes

S. Geffner D. Agrawal A. El Abbadi T. Smith
Department of Computer Science
University of California
Santa Barbara, CA 93106
{sgeffner, agrawal, amr, smithtr}@cs.ucsb.edu

Abstract

Range sum queries on data cubes are a powerful tool for analysis. A range sum query applies an aggregation operation (e.g., SUM) over all selected cells in a data cube, where the selection is specified by providing ranges of values for numeric dimensions. Many application domains require that information provided by analysis tools be current or "near-current." Existing techniques for range sum queries on data cubes, however, can incur update costs on 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. We present an approach that achieves constant time range sum queries while constraining update costs. Our method reduces the overall complexity of the range sum problem.

1 Introduction

The data cube [4], also known in the OLAP community as the multidimensional database [1,8], is designed to provide aggregate information that can be used to analyze the contents of databases and data warehouses. A data cube is constructed from a subset of attributes in the database. Certain attributes are chosen to be *measure attributes*, i.e., the attributes whose values are of interest. Other attributes are selected as *dimensions* or *functional attributes*. The measure attributes are aggregated according to the dimensions. For example, consider a hypothetical database maintained by an insurance company. One may construct a data cube from the database with SALES as a measure attribute, and CUSTOMER_AGE and DATE_OF_SALE as dimensions; such a data cube provides aggregated total sales figures for all combinations of region and date.

Range sum queries are useful analysis tools when applied to data cubes. A range sum query sums the measure attribute within the range of the query; an example is *find the total sales for customers with an age from 37 to 52, over the past three months.* Queries of this form can be very useful in finding trends and in discovering relationships between attributes in the database. Range sum queries over data cubes thus provide a useful tool for analysis. Efficient range-sum querying is becoming more important with the growing interest in database analysis, particularly in On-Line Analytical Processing (OLAP)[2]. Since the introduction of the data cube, there has been considerable research in the database community regarding, for example, choosing subsets of the data cube to precompute [6] and for the processing of aggregation queries [5].

Ho, Agrawal, Megiddo and Srikant [7] have presented an elegant algorithm for computing range queries in data cubes which we call the *prefix sum* approach. The essential idea of the prefix sum approach is to precompute many prefix sums of the data cube, which can then be used to answer ad hoc queries at run-time. The prefix sum method permits the evaluation of any range-sum query on a data cube in constant time. The approach is only hampered by its update cost, which in the worst case requires rebuilding an array of the same size as the entire data cube.

In some problem instances, update cost is not a significant consideration. This is the case, for example, when data is static or is rarely updated. There are, however, applications for which reasonable update cost is important. Many companies are interested in tracking current sales data, for which new information may arrive on a daily basis. As competition increases in the global marketplace, managers demand that their analysis tools provide current or "near-current" information. For such applications, the data cube will fail as a useful analysis tool if it cannot accommodate new information at a reasonable update cost. We note that, as the number of dimensions in a data cube grows, the size of the data cube grows exponentially. Update costs on the order of the size of the data cube may not be practical in these applications, particularly for large data cubes having many dimensions. A method is required that achieves range sum queries in constant time, but that does not require rebuilding the entire data cube during updates.

In this paper, we present a new technique for evaluating range sum queries in data cubes: the *relative prefix sum* approach. This approach achieves constant time

This research is partially supported by NSF grant IRI94-11330.

performance for queries, but constrains update costs. The relative prefix sum approach reduces the overall complexity of the range sum query problem.

Paper Organization The remainder of the paper is organized as follows. In Section 2, we present the model of the range sum problem. In Section 3, we develop the general method used in our algorithms. We introduce the concepts of *overlays* and the *RP array*, and describe their operation. In Section 4, we discuss algorithms for querying and updating these structures. We analyze the performance characteristics of our method, and show that their use reduces the overall complexity of the range-sum problem. We also consider tunable parameters, such as the overlay box size, and discuss other issues relating to performance. Section 5 concludes the paper.

2 The Model

We employ the same model of the range sum problem as presented in [7], which introduces the prefix sum method. Assume the data cube has one measure attribute and d dimensions. Let D={1,2,...,d} denote the set of dimensions. For example, let the measure attribute be SALES, and the dimensions be CUSTOMER_AGE and DATE_OF_SALE. Each dimension has a size n_i, which represents the number of distinct values in the dimension. This size is known a priori, as the number of days in a year, for example, can be assumed to be static. Thus, we can represent the d-dimensional data cube by a d-dimensional array A of size $n_1 \times n_2 \times ... \times n_d$, where $n_i \geq 2$, $i \in D$. In Figure 1, d=2. For clarity, and without loss of generality, our cost model will assume each dimension has the same size; this allows us to present many of the formulas more concisely. Thus, let the size of each dimension be n, i.e. $n=n_1=n_2=\ldots=n_d$. Our subsequent formulae and discussions will refer to n, rather than the total size of the data cube $N=n^d$; in this manner, the impact of the dimensionality of the data cube on performance will be revealed. We will call each array element a *cell*. The total size of array A is n^d cells. We assume the array has starting index 0 in each dimension. For notational convenience, in the two-dimensional examples we will refer to cells in array A as A[i,j], where i is the vertical coordinate and j is the horizontal coordinate. Similarly, we refer to cells in array P as P[i,j], cells in the overlay as O[i,j], and cells in RP as RP[i,j].

Each cell in array A contains the aggregate value of the measure attribute (e.g., total SALES) corresponding to a given point in the d-dimensional space formed by the dimensions. For example, given the measure attribute SALES and the dimensions CUSTOMER_AGE and DATE_OF_SALE, the cell at A[37, 25] contains the total sales to 37-year-old customers on day 25. A range-sum query on array A is defined as the sum of all the cells that fall within the specified range. For example, a range-sum query asking for the total sales to 37-year-old customers from days 20 to 22 would be answered by summing the cells A[37, 20], A[37, 21], and A[37, 22]. We will refer to range-sum queries simply as range queries throughout the rest of this paper. As Ho et. al. point out, the techniques presented here can also be applied to obtain COUNT, AVERAGE, ROLLING SUM, ROLLING AVERAGE, and any binary operator + for which there exists an inverse binary operator - such that a + b - b = a.

We observe the following characteristics of array A. Array A can be used by itself to solve range sum queries; we will refer to this as the *naive* method. Arbitrary range queries on array A can cost $O(n^d)$: a range query over the range of the entire array will require summing every cell in the array. Updates to array A take O(1): given any new value for a cell, an update can be achieved simply by changing the cell's value in the array. Assuming that queries and updates are equally likely, we may represent the overall complexity of a method by taking the product of its query and update costs. For the naive method, this product of query and update costs is $O(n^d) * O(1) = O(n^d)$.

Array A

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

Figure 1. A two-dimensional data cube represented as a two-dimensional array A.

Array P

Index	0	1	2	3	4	5	6	7	8
0	3	8	9	11	13	17	23	26	29
1	10	18	21	29	39	50	57	62	69
2	12	24	29	40	53	67	78	88	102
3	15	29	35	51	67	86	99	117	133
4	19	35	42	61	80	103	123	142	161
5	21	40	50	75	95	126	151	171	191
6	25	49	61	93	114	154	182	205	229
7	27	55	69	103	127	168	205	229	256
8	32	64	81	116	143	186	224	257	290

Figure 2. Array P used in the prefix sum method.

In the prefix sum method, an array P, of the same size as array A, stores various precomputed prefix sums of A (Figure 2). Each cell contains the sum of all cells up to and including itself in array A. For example, cell P[4,0] contains the sum of all cells in the range A[0,0] to A[4,0], or 19, while cell P[2,1] contains the sum of all cells in the range A[0,0] to A[2,1] in array A, or 24. The sum of the entire A array is found in the last cell, P[8,8]. Formally, for all $0 \leq x_i \leq n$ and $i \in D$,

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

Using P, a range query on d dimensions can be answered with a constant (2^d) cell lookups, or O(1). Figure 3 presents the basic idea of the prefix sum method: 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 isolated the region of interest. We note that all such regions begin at cell A[0,0] and extend to some cell in A; thus, the sum of each of these regions can be found directly by reading the value of a single cell in P. The prefix sum method has thus reduced range sum querying to the problem of reading a single individual cell in the P array.

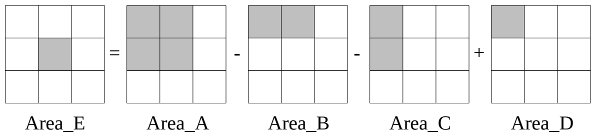

Figure 3. A geometric illustration of the two dimensional case: Sum(Area_E) = Sum(Area_A) - Sum(Area_B) - Sum(Area_C) + Sum(Area_D).

Array A

Index	0	1	2	3	4	5	6	7	8
0	3	5	1	2	2	4	6	3	3
1	7	* 4	2	6	8	7	1	2	4
2	2	4	2	3	3	3	4	5	7
3	3	2	1	5	3	5	2	8	2
4	4	2	1	3	3	4	7	1	3
5	2	3	3	6	1	8	5	1	1
6	4	5	2	7	1	9	3	3	4
7	2	4	2	2	3	1	9	1	3
8	5	4	3	1	3	2	1	9	6

Array P

Index	0	1	2	3	4	5	6	7	8
0	3	8	9	11	13	17	23	26	29
1	10	* 19	22	30	40	51	58	63	70
2	12	25	30	41	54	68	79	89	103
3	15	30	36	52	68	87	100	118	134
4	19	36	43	62	81	104	124	143	162
5	21	41	51	76	96	127	152	172	192
6	25	50	62	94	115	155	183	206	230
7	27	56	70	104	128	169	206	230	257
8	32	65	82	117	144	187	225	258	291

Figure 4. Update example, prefix sum method.

While this method provides constant time queries, in the worst case it incurs update costs proportional to the entire data space. This update cost results from the very dependencies in the data that allow the method to work. As noted, the values of cells in array P are cumulative, in that they contain the sums of all cells in array A that precede them. Thus, when a cell in A is updated, all cells in P that follow it will also require updating. Figure 4 shows the array P as the cell A[1,1] is about to be updated. The value of cell A[1,1] is a component of every cell in the shaded region; thus, updating A[1,1] requires updating every P cell in the shaded region. In the worst case, when cell A[0,0] is updated, this cascading update property will require that every cell in the data cube be updated. Since the size of the data cube is n^d cells, the resulting update complexity is $O(n^d)$. The prefix sum solution has thus traded update and lookup characteristics with the naive method. The product of the query and update costs for the prefix sum method is O(1) * $O(n^d)$ = $O(n^d)$, which is identical to the naive method.

3 The Relative Prefix Sum Method

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, updates to its data structure are very expensive, due to the cascading updates effect. As noted earlier, some application domains require updates on a regular basis; when data cubes are very large, and have many dimensions, an update cost of $O(n^d)$ can become impractical. We seek a method that has the query power of the prefix sum approach, but improves its update performance.

The expensive update cost in P is a result of the dependencies between values in its cells. These dependencies in turn cause the cascading update effect. We cannot, however, completely eliminate these dependencies. The prefix sum method works precisely because of the dependencies between its cells. In general, when we eliminate dependencies we pay for it with an increased query cost; if we eliminate all dependencies, we will be left with the naive approach, whose query cost is $O(n^d)$. Since we cannot completely eliminate dependencies, we also cannot completely eliminate cascading updates. However, perhaps we can create boundaries that limit cascading updates to distinguished cells. In doing so, we must be careful not to introduce query overhead that results in the loss of the constant-time query characteristic.

The relative prefix sum approach is a new method for answering range sum queries on data cubes. The method adds and subtracts region sums to obtain the complete sum of the query region, as illustrated in Figure 3. The method makes use of new data structures that provide constant time queries while controlling cascading updates. By creating boundaries that limit cascading updates to distinguished cells, the method reduces the overall complexity of the range sum problem.

Our method makes use of two components: an *overlay* and a *relative-prefix* (RP) array. An overlay partitions array A into fixed size regions called *overlay boxes*. Overlay boxes store information regarding the sums of regions of array A preceding them. RP is an array of the same size as array A; it contains relative prefix sums within regions defined by the overlay. Using the two components in concert, we will construct prefix sums "on the fly". Together, the components limit cascading updates to distinguished cells. We first describe overlays, then describe RP.

3.1 Overlays

We define an *overlay* as a set of disjoint hyperrectangles (hereafter called "boxes") of equal size that completely partition array A into equal sized regions of cells (Figure

5). In the figure, array A has been partitioned into overlay boxes of size 3×3. For clarity, and without loss of generality, let the length of the overlay box in each dimension be k. The size of array A is n^d, thus the total number of overlay boxes is $\left\lceil \frac{n}{k} \right\rceil^d$.

We define several terms for use later in the paper. We say that an overlay box is *anchored at* $(a_1, a_2, ..., a_d)$ if the box corresponds to the region of array A where the first cell (lowest cell index in each dimension) is $(a_1, a_2, ..., a_d)$; we denote this overlay box as $B[a_1, a_2, ..., a_d]$. The first overlay box is anchored at (0, 0, ..., 0). An overlay box $B[a_1, a_2, ..., a_d]$ is said to *cover* a cell $(x_1, x_2, ..., x_d)$ in array A if the cell falls within the boundaries of the overlay box, i.e., if $\forall i((a_i \le x_i) \wedge (a_i + k \ge x_i))$.

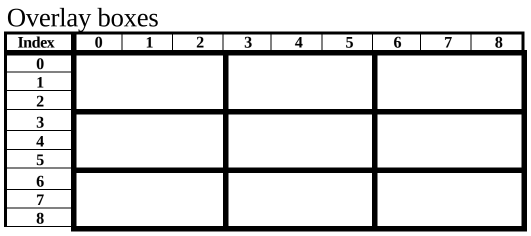

Figure 5. Array A partitioned into boxes of size 3x3.

Refer to Figure 5. In the figure, k=3; i.e., each overlay box in the figure is of size 3×3. The total number of overlay boxes is $\left\lceil \frac{n}{k} \right\rceil^d = (9/3)^2 = 9$. The boxes are anchored at cells (0,0), (0,3), (0,6), (3,0), (3,3), (3,6), (6,0), (6,3), and (6,6). Each overlay box corresponds to an area of array A of size k^d cells; thus, in this example each overlay box covers $3^2 = 9$ cells of array A.

V	X_1	X_2
Y_1		
Y_2		

Figure 6. Values stored in an overlay box of size 3x3.

Figure 6 shows the values stored in an overlay box. Each overlay box stores an *anchor value*, plus $(k^d - (k-1)^d)-1$ *border values*. V is the anchor cell, while X_1, X_2 and Y_1, Y_2 are border cells. The other cells covered by the overlay box are not needed in the overlay, and would not be stored.

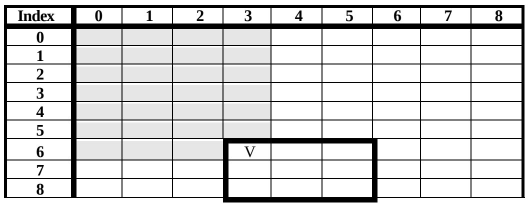

Figure 7. Array A showing calculation of overlay box anchor.

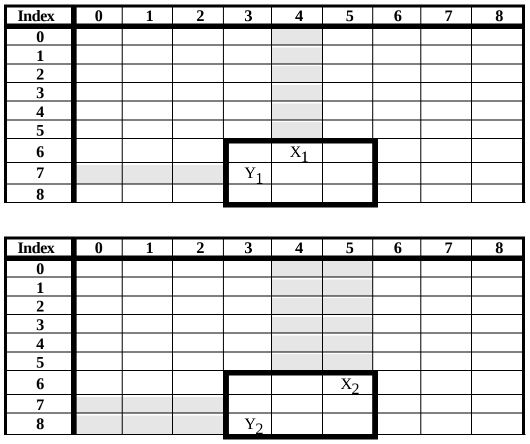

Figure 8. Array A showing calculation of overlay box border values.

Values stored in an overlay box provide sums of regions outside the box. The anchor value is the sum of all cells in A up to, but not including, the cell under V. Figure 7 shows an overlay box superimposed on array A. The anchor value of the overlay box is equal to the sum of the shaded region in array A. More formally, an overlay box anchored at $(a_1, a_2, ..., a_d)$ has an anchor value that is equal to $SUM(A[0, 0, ..., 0]:A[a_1, a_2, ..., a_d]) - A[a_1, a_2, ..., a_d]$. Figure 8 shows the calculation of border values. The border values are equal to the sum of the associated shaded regions of array A. Border value 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. Border value Y_1 is the sum of all cells in the row to the left of the cell containing Y_1. Border value Y_2 is the sum of all cells in the row to the left of its cell, plus the cells to the left of Y_1. Note that border values are cumulative (i.e., X_2 includes the value of X_1, and in general X_n includes the values of $X_1..X_{n-1}$). Border values for other dimensions are calculated in the same manner. Formally, the border value contained in cell $[i_1, i_2, ..., i_d]$ is equal to $SUM(A[0, 0, ..., 0]:A[i_1, i_2, ..., i_d]) - RP[i_1, i_2, ..., i_d]$ - the anchor value of the overlay box.

As noted earlier, the regions used in Figure 3 all begin at A[0,0] and end at an arbitrary cell in A (the *target cell*). Figure 9 shows such a target cell, marked as *; for reference, the overlay box covering the cell has been superimposed on array A. In this example, the cell is located at (7,5). We wish to determine the sum of the region that begins at A[0,0] and ends at A[7,5]. We will calculate this region sum in two steps. The first step uses the overlay; the second step uses RP. Using values in the overlay, we first calculate the sum of the shaded area in Figure 9. The shaded area represents the portion of the desired region that lies outside the overlay box. To determine the sum of this area, we add border values Y_1

and X_2 to the anchor value. The anchor value provides the sum of all cells from A[0,0]:A[6,3], minus cell A[6,3]. Border value Y_1 provides the sum of the cells in A[7,0]:A[7,2]. Border value X_2 provides the sum of cells in A[0,3]:A[5,5]. These respective regions are outlined in the figure. Adding the anchor value and two border values thus provides the sum of all cells in the shaded region.

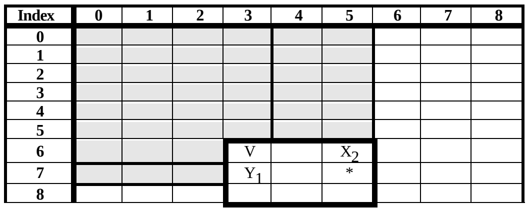

Figure 9. Addition of border values Y1 and X2 with anchor value V.

3.2 Relative Prefix Array (RP)

The relative prefix array (RP) is the same size as array A. It is partitioned into regions of cells that correspond to overlay boxes. Figure 10 shows RP with overlay boxes drawn for reference. Each region in RP contains prefix sums that are relative to the area enclosed by the box. Each region of RP is independent of other regions. Figure 11 shows an area of RP corresponding to one overlay box. In the figure, * denotes the location of a cell within the box. The value of cell * is the sum of the cells in array A that fall within the shaded region. More formally, given a cell RP[i_1, i_2, ..., i_d] and the anchor cell location (v_1, v_2, ..., v_3) of the overlay box covering this cell, the value stored in RP[i_1, i_2, ..., i_d] is SUM(A[v_1, v_2, ..., v_3]:A[i_1, i_2, ..., i_d]).

Relative Prefix (RP) array

Index	0	1	2	3	4	5	6	7	8
0	3	8	9	2	4	8	6	9	12
1	10	18	21	8	18	29	7	12	19
2	12	24	29	11	24	38	11	21	35
3	3	5	6	5	8	13	2	10	12
4	7	11	13	8	14	23	9	18	23
5	9	16	21	14	21	38	14	24	30
6	4	9	11	7	8	17	3	6	10
7	6	15	19	9	13	23	12	16	23
8	11	24	31	10	17	29	13	26	39

Figure 10. RP array with overlay boxes drawn for reference.

As noted earlier, the overlay values provide a portion of the complete region sum. RP provides the final portion of that sum. Figure 12 shows the construction of a complete region sum using the overlay and RP. In the figure, * denotes an arbitrary cell in A; here, the cell is A[7,5]. For reference, the overlay box covering this cell has been superimposed on array A. The anchor value and border values from the overlay box provide the sum of the portion of the shaded region outside the overlay box as noted above. The cell * in RP provides the sum of the portion of the shaded region within the overlay box. Adding the two sums together yields the sum of all cells in array A that fall within the shaded region. In this manner, the overlay and RP can be used to generate any region sum rooted at A[0,0, ..., 0], and they are therefore sufficient to provide the region sums required by the method illustrated in Figure 3. This process requires one anchor value, d border values, and one value from RP.

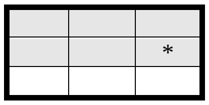

Figure 11. Area of RP corresponding to one overlay box.

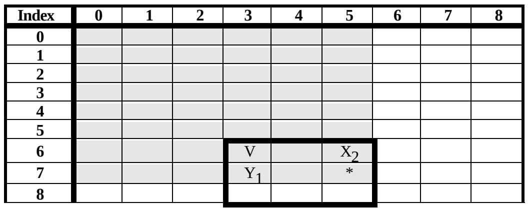

Figure 12. Construction of complete region sum from overlay and RP.

3.3 Examples Using the Overlay and RP

We now present several concrete examples using an overlay and RP. Figure 13 shows array A, the overlay and RP. For reference, the overlay partitions have been drawn on RP. The following examples refer to this figure. We first provide examples of calculating anchor and border values. Finally, we find a complete region sum.

Calculating Anchor and Border Values We will calculate the anchor and border values for the overlay box anchored at cell (3,3). The anchor value in overlay cell O[3,3] is equal to SUM(A[0,0]..A[3,3]) - A[3,3] = 51-5 = 46. This is the sum of all cells in A up to, but not including, cell A[3,3]. The border value in overlay cell [4,3] = SUM(A[0,0]..A[4,3]) - RP[4,3] - anchor[3,3], or 61-8-46=7. The border value in overlay cell [5,3] = SUM(A[0,0]..A[5,3]) - RP[5,3] - anchor[3,3], or 75-14-46=15. Similarly, the border value in overlay cell [3,4] = SUM(A[0,0]..A[3,4]) - RP[3,4] - anchor[3,3], or 67-8-46=13. The border value in overlay cell [3,5] = SUM(A[0,0]..A[3,5]) - RP[3,5] - anchor[3,3], or 86-13-46=27.

Calculating a Complete Region Sum Suppose we wish to determine the sum of the cells in array A from cell A[0,0] to cell A[7,5]. Cell (7,5) is covered by the overlay box anchored at cell (6,3). Let us first find the sum of the portion of the shaded area that falls outside the overlay box; this requires an anchor value plus two border values. The anchor value of the overlay box is 86. We now require two border values. Cell (7,5) is one cell down from, and two cells to the right of, the anchor cell; thus, we need border values Y_1 and X_2. Border value Y_1 is found in overlay cell O[7,3], and has the value 8. Border value X_2 is found in overlay cell O[6,5], and has the value 51. Thus, the shaded area outside the overlay box has a value equal to 86+51+8. We now find the sum of the cells in the portion of the shaded area inside the overlay box. This sum is found in RP[7,5]; its value is 23. The complete region sum for the region A[0,0]:A[7,5] is thus 86+51+8+23=168.

Array A

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

Overlay boxes of size 3x3

Index	0	1	2	3	4	5	6	7	8
0	0	0	0	9	0	0	17	0	0
1	0			12			33		
2	0			20			50		
3	12	12	17	46	13	27	97	10	24
4	0			7			17		
5	0			15			40		
6	21	19	29	86	20	51	179	20	40
7	0			8		*	14		
8	0			20			32		

Relative Prefix (RP) array

Index	0	1	2	3	4	5	6	7	8
0	3	8	9	2	4	8	6	9	12
1	10	18	21	8	18	29	7	12	19
2	12	24	29	11	24	38	11	21	35
3	3	5	6	5	8	13	2	10	12
4	7	11	13	8	14	23	9	18	23
5	9	16	21	14	21	38	14	24	30
6	4	9	11	7	8	17	3	6	10
7	6	15	19	9	13	* 23	12	16	23
8	11	24	31	10	17	29	13	26	39

Figure 13. Arrays used in the examples.

4 Operations Using the Approach

Algorithms for query and update using the approach are found in [3]. We present a brief discussion of the algorithms here.

4.1 Range Sum Queries

The algorithm for range sum queries calculates 2^d region sums, as illustrated in Figure 3. Calculating each region sum requires adding one anchor value, d border values, and one value from RP. We assume the number of dimensions for a given data cube is fixed, as in [7]. Since the sum of each region can be obtained in constant time, and a constant number of region sums are needed to answer range sum queries on the data cube, our method provides constant-time query performance.

4.2 Updates

Updating a cell c in the data cube requires updates to RP and the overlay. Updates in RP are constrained to a region covered by one overlay box; RP cells within the overlay box boundary that are greater in index than the updated cell are affected. Updating the overlay requires several steps. Changes to the overlay fall into several categories, as shown in Figure 14. In the figure, * marks the location of the changed cell. There is one group of overlay boxes per dimension whose border values may need updating. The shaded region to the right of the changed cell shows the overlay boxes whose border values in the first dimension will be affected by the update. The shaded region below the changed cell shows the overlay boxes whose border values in the second dimension will be affected by the update. The non-shaded region below and to the right of the changed cell contains overlays whose anchor cells alone will be affected by the update; we refer to these overlay boxes as being *interior* to the update. Affected border values for each dimension are updated in turn; note that only border values in columns greater than or equal to * are affected by the update. Finally, anchor values of interior overlay boxes are updated.

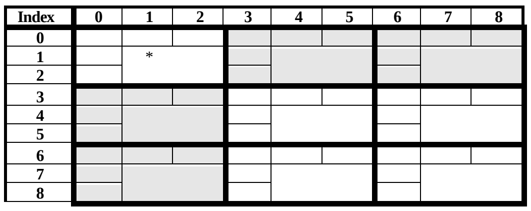

Figure 14. Overlay regions during an update.

Relative Prefix (RP) array

Index	0	1	2	3	4	5	6	7	8
0	3	8	9	2	4	8	6	9	12
1	10	* 19	22	8	18	29	7	12	19
2	12	25	30	11	24	38	11	21	35
3	3	5	6	5	8	13	2	10	12
4	7	11	13	8	14	23	9	18	23
5	9	16	21	14	21	38	14	24	30
6	4	9	11	7	8	17	3	6	10
7	6	15	19	9	13	23	12	16	23
8	11	24	31	10	17	29	13	26	39

Overlay boxes

Index	0	1	2	3	4	5	6	7	8
0	0	0	0	9	0	0	17	0	0
1	0	*		13			34		
2	0			21			51		
3	12	13	18	47	13	27	98	10	24
4	0			7			17		
5	0			15			40		
6	21	20	30	87	20	51	180	20	40
7	0			8			14		
8	0			20			32		

Figure 15. Update example, relative prefix sum method.

Update Example Figure 15 shows an update to the overlay and RP. * denotes the location of the changed cell in array A. Recall that each region of RP is independent. Only RP-cells in the same overlay box as the changed cell will be modified; in this case, the four shaded cells are modified. Updates cascade in RP within the overlay box boundary, but cascading stops at the boundary; cells in RP covered by other overlay boxes will not be modified. In addition, cells in several groups of overlay boxes are affected. Overlay boxes directly to the right of the change will require updates to border values in rows greater than or equal to the changed cell (e.g., overlay cells [1,3],[2,3] and [1,6],[2,6]). Overlay boxes directly below the change will require updates to all border values in columns greater than or equal to the changed cell (e.g., overlay cells [3,1],[3,2] and [6,1],[6,2]). Interior overlay boxes will have only their anchor values changed (e.g., overlay cells [3,3], [3,6], [6,3] and [6,6]). In this example, twelve overlay cells are modified. Note that when an update occurs to a cell directly under an anchor cell, e.g. cell [0,0], this would require only updating anchor cells in other overlay boxes; no border values would then need to be changed in any overlay box.

The relative prefix sum method limits cascading updates to distinguished cells. In the example, the total update cost for the overlay algorithm is sixteen cells (twelve overlay cells and four cells in RP), compared to sixty four cells in the prefix sum method (Figure 4). In the next section of the paper, we examine performance characteristics and tunable parameters of the relative prefix sum method.

4.3 Choosing the Overlay Box Size

As noted earlier, finding a value in P using RP and the overlay requires one anchor value, d border values and one value from RP. Range sum queries using the overlay box method are thus achieved in constant time. This is irrespective of the overlay box size.

However, the overlay box size does affect update costs. Assume a data cube with d dimensions, each dimension having size n, and an overlay box of size k on each dimension. For convenience, assume n is evenly divisible by k. Further assume that the cost of writing an overlay cell is equal to the cost of writing a cell in RP, as is the case when both structures are in main memory (RAM). In the worst case, an update to the data cube will affect $(k-1)^d$ cells in the RP array + $d(n/k)(k^{d-1})$ overlay border cells + $(n/k-1)^d$ overlay anchor cells. This formula can be reasonably approximated as

$$k^d + d(n/k)(k^{d-1}) + (n/k)^d$$

which can be simplified as

$$k^d + dnk^{d-2} + (n/k)^d.$$

In the formula, note that the term k^d accounts for the total cost of updates in RP; the remaining costs are associated with updating the overlay. We are considering the case where the cost of updating cells in the overlay and cells in RP are identical; therefore, we can minimize the total update cost by minimizing the equation as a whole.

By using approximation, we find that the cost is minimized when the overlay box size is chosen to be

$$k=\sqrt{n}.$$

When the box size is larger, fewer overlay cells will be updated, but more cells in RP will be updated, leading to a larger total number of affected cells. When the box size is smaller, fewer cells in RP are updated, but more cells in the overlay are updated, again resulting in a larger total number of affected cells. Choosing a box size of $\sqrt{n}$ results in the lowest total number of cells in overlays and RP being changed. This is desirable when the cost of accessing cells in overlays is equal to the cost of accessing cells in RP. With a box size of $k=\sqrt{n}$, the worst-case update formula yields $O(n^{d/2})$.

The product of the query cost and update cost is thus $O(1) * O(n^{d/2}) = O(n^{d/2})$. This is in contrast to the prefix sum algorithm and the naive method, both of which have a total cost of $O(n^d)$. The relative prefix sum approach thus reduces the overall cost of the problem.

4.4 Practical Considerations

In many practical applications, the large size of RP would require that it be stored on disk. In this case, it is possible that overlays can be maintained in main memory, even when RP is very large. Recall that k is the length of the overlay box in each dimension. As noted earlier, the overlay box covers an area of array RP equal to k^d cells. Each overlay box requires $(k^d - (k-1)^d)$ cells of storage. The storage required by the overlay is thus significantly smaller than that used by the corresponding area of array RP, and space savings grow larger as the size of the overlay box grows. For example, consider a two dimensional array RP and an overlay size of 100×100 cells. The overlay box needs $(100^2 - 99^2)=199$ cells of storage, while the region of RP covered by the overlay box requires 10,000 cells; the overlay box requires less than 2% of the storage of the corresponding region of RP. Given suitable box sizes, it may be feasible to keep all of the overlay boxes in main memory, while RP resides on disk. Figure 16 shows the storage requirements of overlay boxes as a percentage of the storage required by the region they cover in RP as d and k are varied. Note that, as the overlay box size grows, overlay boxes use dramatically less storage than the region in RP they cover.

Maintaining overlays in main memory would have a substantial positive impact on query and update performance. In particular, much of the method's update cost is incurred in updating overlay box values. The optimal box size formula assumes that the cost of accessing overlay cells is the same as the cost of accessing cells in RP; but when overlays are in memory, their access cost is much smaller than the cost of accessing cells in RP on disk. Consequently, we can expect the optimal overlay box size in this configuration to be larger than the formula predicts. Larger box sizes are desirable because they save more storage, and result in fewer overlay boxes being modified during the update process. Since disks are block-based devices, the cost of accessing a cell in RP is related to the cost of accessing a disk block. In this configuration, it would be preferred to set the overlay box size such that the corresponding region of RP fits exactly into a constant number of disk pages; both queries and updates will then require only a constant number of disk reads or writes.

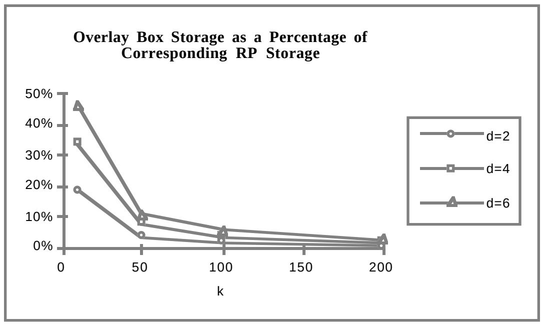

Figure 16. Comparison of overlay and RP storage requirements as d and k are varied.

When both the overlay and RP are stored on disk, one could still expect the optimal overlay box size to be somewhat larger than the formula predicts. Since overlay boxes in general require less storage than the area of RP they cover, they can presumably be packed more efficiently into disk blocks. Recall that updates only effect one region of RP, but potentially many overlay boxes. When overlay boxes are packed efficiently into disk blocks, the aggregate cost of accessing them should be less than the cost of accessing a region in RP.

5 Conclusion

Efficient calculation of range sum queries has become more important in recent years, owing to an increasing demand for OLAP and data cube applications. Many business applications require that data cubes be updated with current information on a regular basis. For large data cubes that are updated weekly or daily, the performance characteristics of the naive approach and the prefix sum approach may be insufficient. The naive method requires $O(n^d)$ for queries, which is clearly undesirable, but requires only O(1) for updates. The prefix sum method trades costs with the naive method, having O(1) query cost and $O(n^d)$ update cost; however, for large data cubes, having many dimensions, this update cost may place a severe limit on the frequency of updates. The product of the query and update costs for both the naive method and the prefix sum method is $O(n^d)$.

The relative prefix sum method provides constant time evaluation of range sum queries. The method constrains the cascading update problem encountered by the prefix sum approach; as a result, its update complexity is reduced to $O(n^{d/2})$. The product of the query and update costs for the relative prefix sum approach is $O(n^{d/2})$. Thus, the relative prefix sum method not only reduces update costs when compared to the prefix sum method, but it also reduces the overall complexity of the range sum problem.

References

[1] R. Agrawal, A. Gupta, S. Sarawagi. Modeling multidimensional databases. In *Proc. of the 13th Int'l Conference on Data Engineering*, Birmingham, U.K., April 1997.

[2] E. F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: an IT mandate. Technical report, E.F. Codd and Associates, 1993.

[3] S. Geffner, D. Agrawal, A. El Abbadi, T. Smith. Algorithms for the Relative Prefix Sum Approach to Range Sum Queries in Data Cubes. University of California, Santa Barbara, Computer Science Technical Report TRCS99-01. At http://www.cs.ucsb.edu.

[4] J. Gray, A. Bosworth, A. Layman, H. Pirahesh. Data Cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. In *Proc. of the 12th Int'l Conference on Data Engineering*, pages 152-159, 1996.

[5] A. Gupta, V. Harinarayan, D. Quass. Aggregate-query processing in data warehousing environments. In Proc. of the Eighth Int'l Conference on Very Large Databases (VLDB), pages 358-369, Zurich, Switzerland, September 1995.

[6] V. Harinarayan, A. Rajaraman, J. D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD Conference on the Management of Data*, June 1996.

[7] C. Ho, R. Agrawal, N. Megiddo, R. Srikant. Range Queries in OLAP Data Cubes. In *Proc. of the ACM SIGMOD Conference on the Management of Data*, pages 73-88, 1997.

[8] The OLAP Council. *MD-API the OLAP Application Program Interface Version 5.0 Specification*, September 1996.