Data Cubes in Dynamic Environments

Steven P. Geffner   Mirek Riedewald   Divyakant Agrawal   Amr El Abbadi

Department of Computer Science
University of California, Santa Barbara, CA 93106 *

Abstract

The data cube, also known in the OLAP community as the multidimensional database, is designed to provide aggregate information that can be used to analyze the contents of databases and data warehouses. Previous research mainly focussed on strategies for supporting queries, assuming that updates do not play an important role and can be propagated to the data cube in batches. While this might be sufficient for most of today's applications, there is growing evidence that modern interactive data analysis applications will have to balance update and query costs. Two techniques for maintaining data cubes in dynamic environments are described here. The first, Relative Prefix Sums *(RPS), supports a constant response time for ad-hoc range sum queries on the data cube, while at the same time greatly reducing the update costs compared to prior approaches. The second, the* Dynamic Data Cube *(DDC), guarantees a sub-linear cost for both range sum queries and updates.*

1 Introduction

A data cube or multidimensional database ([7] [4] [1]) 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 age 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 to "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. Efficient range-sum querying is becoming more important with the growing interest in database analysis, particularly in On-Line Analytical Processing (OLAP) [3].

Ho et al. [8] have presented an elegant algorithm for computing range sum queries in data cubes which we call the *Prefix Sum* (PS) approach. The essential idea is to precompute many prefix sums of the data cube, which can

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was partially supported by NSF grants IIS 98-17432 and IIS 99-70700

A	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

P	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 1: The original array (A, left) and the cumulative array used for the Prefix Sum method (P, right)

then be used to answer ad hoc queries at run-time (see Figure 1). The Prefix Sum method permits the evaluation of any range-sum query on a data cube in constant time. The approach is mainly hampered by its update cost, which in the worst case requires rebuilding an array of the same size as the entire data cube. [6] describes a technique that considerably improves the update performance compared to PS, but still provides a constant range sum query cost. The main idea is to control the cascading updates. The Hierarchical Cubes techniques [2] generalize this idea by offering the user different tradeoffs between update and query cost. The only technique that provides guaranteed sub-linear update and range query cost is the Dynamic Data Cube [5].

In some problem instances, update cost is not a significant consideration. There are, however, many current and emerging applications for which reasonable update cost becomes important. In Section 2 we discuss some dynamic scenarios. Then in Section 3 the Relative Prefix Sum and the Dynamic Data Cube techniques are presented. Section 4 concludes this article.

2 Why Do Updates Matter?

Update complexity is often considered to be unimportant in current-day data analysis applications. These systems are oriented towards batch updates, and for a wide variety of business applications this is considered sufficient. Nevertheless, the batch updating paradigm, a holdover from the computing environment of the 1960's, is tremendously limiting to the field. The Prefix Sum method is a good example of present-day cutting-edge data cube technology. Any range sum query can be answered in constant time. During updates, however, it requires in the worst case updating an array whose size is equal to the size of the entire data cube. It is easy to see that, even under batch update conditions, this model is not workable for many emerging applications (e.g., what if the size of the data cube were a terabyte?).

There is no doubt that OLAP applications typically have to deal with updates. Data warehouses collect constantly changing data from a company's databases; digital libraries and data collections grow with an increasing rate. But isn't it good enough if those updates can be efficiently processed in batches? Why instantly propagating each update to the data cube? Why not just collect all updates during the day and then apply them overnight when nobody uses the data collection? There are several arguments.

- For some applications, it is desirable to incorporate updates as soon as possible. Batch updating unnecessarily limits the range of choices. While some applications do not suffer in the presence of stale data, in many emerging applications, e.g. decision support and stock trading, the instant availability of the latest information plays a crucial role.

- OLAP means *interactive* data analysis. For instance, business leaders will want to construct interactive what-if scenarios using their data cubes, in much the same way that they construct what-if scenarios using spreadsheets now. These applications require real-time (or even hypothetical) data to be integrated with

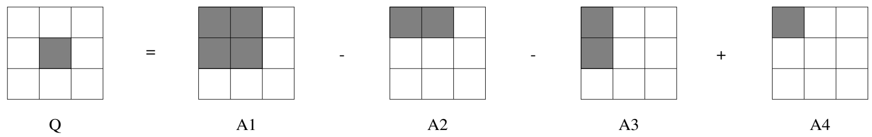

Figure 2: A geometric illustration of the two-dimensional case: SUM(Q) = SUM(A1) - SUM(A2) - SUM(A3) + SUM(A4)

historical data for the purpose of instantaneous analysis and subsequent action. The fact that there are significant impediments to updates in popular data cube techniques prevents these and many other emerging applications from being deployed.

- Batch updates incur another serious handicap. Even though the *average* cost per update might be small, performing the complete batch of updates takes a considerable amount of time. During this time the data in the data cube is generally not accessible to an analyst. The greater the amount of updates, the worse the situation can get. Finding a suitable time slot for this *update window* becomes increasingly harder when businesses demand flexible work hours and 24 hour availability of their data. Also, data collections that are accessible from all over the world (e.g., for multinational companies) do not follow the simple "many accesses during daytime, no accesses at nighttime" pattern.

By reducing the barriers to frequent updates in very large data cubes, new and interesting applications become possible; traditional applications can profit from greater flexibility and 24 hour availability of the data. With growing data collections and a growing demand for interactive analysis of up to date data, traditional approaches like batch updates and re-computation of the complete data cube can not be regarded as sufficient any more for an increasing number of applications.

3 Two Dynamic Data Cube Approaches

In the following, two data cube techniques for dynamic environments are presented. Compared to the Prefix Sum technique [8] they trade query efficiency for faster updates. Both make use of the inverse property of addition by adding and subtracting region sums to obtain the complete sum of the query region, in the same manner as the PS method. As Ho et al. point out, the technique can be applied to any operator $\oplus$ for which there exists an inverse operator $\ominus$ such that $a \oplus b \ominus b = a$ (e.g., COUNT, AVERAGE, ROLLING SUM, ROLLING AVERAGE).

Let A denote the original array, d its dimensionality and n the number of possible indexes (attribute values) for each dimension[1]. $(y_1, \ldots, y_d)$ describes a single *cell*, i.e., a point of the multidimensional data space. Without loss of generality we assume that $(0, \ldots, 0)$ is the point of the array with the smallest index in each dimension. Then the sum for an arbitrary range query can be obtained as the result of combining (adding/subtracting) up to 2^d range sums of the form $\text{SUM}(A[0, \ldots, 0] : A[x_1, \ldots, x_d])$. Figure 2 illustrates the calculations for a two-dimensional data cube. With $\text{SUM}(A[y_1, \ldots, y_d] : A[z_1, \ldots, z_d])$ we refer to the aggregate sum of all cells enclosed in the bounding box described by $(y_1, \ldots, y_d)$ ("upper left" corner) and $(z_1, \ldots, z_d)$ ("lower right" corner). The "upper left" corner of a hyper-rectangle, i.e., the point of the rectangle with the smallest index in each dimension, will be referred to as the *anchor* of that hyper-rectangle. Since an arbitrary range sum can be obtained as the combination of up to 2^d (which is independent of the size of the range) range sums for ranges anchored at $(0, \ldots, 0)$, we will only focus on handling those ranges.

[1]Choosing a single parameter n is done for the sake of clarity and results in simpler formulas. Our techniques, however, are not restricted to data cubes with domains of equal sizes.

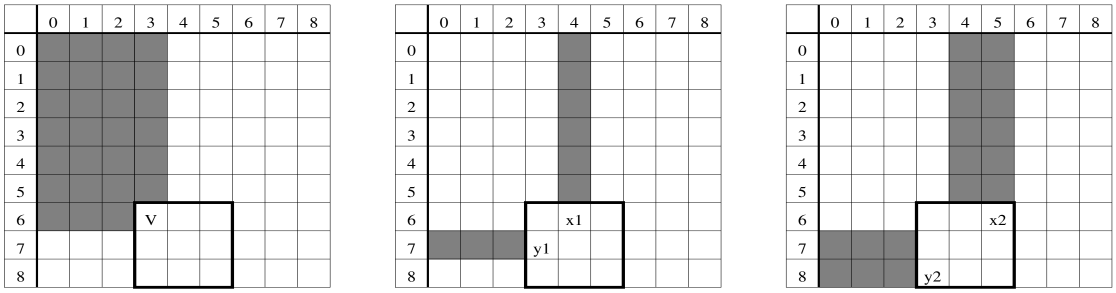

Figure 3: Calculation of overlay values as the sum of the cells in the shaded area on array A

3.1 The Relative Prefix Sum Approach

The Relative Prefix Sum (RPS) method [6] provides constant-time queries with reduced update complexity (compared to the Prefix Sum technique), and is suitable for applications where constant time queries are vital but updates are more frequent than the Prefix Sum method will allow. The main idea behind RPS is to control the cascading updates that lead to poor update behavior.

The RPS method makes use of two components: an *overlay* (*OL*) and a *relative-prefix* (*RP*) array. The 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* contains relative prefix sums within regions defined by the overlay. Using the two components in concert, we construct prefix sums "on the fly". We first describe the overlay, then describe *RP*.

3.1.1 Overlay

We define an overlay as a set of disjoint hyper-rectangles of equal size, further on called *overlay boxes*, that completely partition array A into regions of cells. 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 $\lceil n/k \rceil^d$. The first overlay box is anchored at $(0, \ldots, 0)$. Let overlay box B be anchored at $(b_1, \ldots, b_d)$. B 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 overlay box, i.e., if for all i: $b_i \leq x_i < b_i + k$. A single cell o of an overlay box *aggregates* a cell a of array A outside the overlay box, if the value of o depends on a's value. Each overlay box corresponds to an area of array A of size k^d cells. The values stored in an overlay box provide sums of regions outside the box. In the two-dimensional example in Figure 3 the cells in the top row and the leftmost column contain the sums of the values in the corresponding shaded cells of array A (those overlay cells aggregate the respective cells in the shaded area). The other cells covered by the overlay box are not needed in the overlay, and would not be stored.

In general only overlay cells in the "upper left" surfaces are needed, i.e., in those surfaces that contain the anchor cell. More formally, overlay box B anchored at $(b_1, \ldots, b_d)$ aggregates k^d overlay cells $O = (o_1, \ldots, o_i, \ldots, o_d)$, namely those cells that satisfy for each dimension i: $b_i \leq o_i < b_i + k$. Among those overlay cells, only $k^d - (k-1)^d$ are used, namely those where a dimension j exists, such that $o_j = b_j$ (compare to the two-dimensional example). The anchor cell $(b_1, \ldots, b_d)$ stores the value

$$(\sum_{a_1=0}^{b_1} \cdots \sum_{a_d=0}^{b_d} A[a_1, \ldots, a_d]) - (\sum_{a_1=b_1}^{b_1} \cdots \sum_{a_d=b_d}^{b_d} A[a_1, \ldots, a_d])$$

(the sum of all cells in $A[0, \ldots, 0] : A[b_1, \ldots, b_d]$ excluding $(b_1, \ldots, b_d)$). For an overlay cell O where for exactly one dimension j $o_j > b_j$ and for all other dimensions i $o_i = b_i$ the value is calculated as

$$(\sum_{a_1=0}^{b_1} \cdots \sum_{a_{j-1}=0}^{b_{j-1}} \sum_{a_j=b_j+1}^{o_j} \sum_{a_{j+1}=0}^{b_{j+1}} \cdots \sum_{a_d=0}^{b_d} A[a_1, \ldots, a_d])$$

$$-(\sum_{a_1=b_1}^{b_1} \cdots \sum_{a_{j-1}=b_{j-1}}^{b_{j-1}} \sum_{a_j=b_j+1}^{o_j} \sum_{a_{j+1}=b_{j+1}}^{b_{j+1}} \cdots \sum_{a_d=b_d}^{b_d} A[a_1, \ldots, a_d])$$

In general the value stored in a used overlay cell $O = (o_1, \ldots, o_d)$ is

$$(\sum_{a_1=l_1}^{u_1} \cdots \sum_{a_d=l_d}^{u_d} A[a_1, \ldots, a_d]) - (\sum_{a_1=m_1}^{v_1} \cdots \sum_{a_d=m_d}^{v_d} A[a_1, \ldots, a_d])$$

where for all dimensions i:

- if $o_i = b_i$: $l_i = 0, u_i = b_i, m_i = b_i, v_i = b_i$
- if $o_i > b_i$: $l_i = b_i + 1, u_i = o_i, m_i = b_i + 1, v_i = o_i$

Intuitively the first sum includes all cells that fall into the hyper-rectangle $A[0, \ldots, 0] : A[o_1, \ldots, o_d]$, and that are not aggregated by another overlay cell whose coordinates can be obtained by replacing some of the o_i that are greater than b_i by b_i (e.g., in Figure 3 V's coordinates $(3, 6)$ can be obtained by replacing x_2's x-coordinate 5 with the corresponding overlay anchor coordinate 3, therefore x_2's summation in x-direction has to range from (3+1) to 5). The second term subtracts the values of those cells that fall into overlay box B and should therefore not be included in the summation (in Figure 3 for x_2 the values in cells $(4, 6)$ and $(5, 6)$ have to be subtracted).

3.1.2 Relative Prefix Array (*RP*)

The relative prefix array (*RP*) is of the same size as array A. It is partitioned into regions of cells that correspond to overlay boxes. Each region in *RP* contains prefix sums that are relative to the area enclosed by the box, i.e., it is independent of other regions. More formally, given a cell *RP*$[i_1, \ldots, i_d]$ and the anchor cell location $(v_1, \ldots, v_d)$ of the overlay box covering this cell, the value stored in *RP*$[i_1, \ldots, i_d]$ is $\text{SUM}(A[v_1, \ldots, v_d] : A[i_1, \ldots, i_d])$.

3.1.3 Query and Update Operations

The range sum for any query anchored at $(0, \ldots, 0)$ can be obtained by adding the corresponding values stored in the overlay and in *RP*. The overlay values and *RP* are therefore sufficient to provide the region sums required by the method illustrated in Figure 2. Figure 4 shows the two data structures for our example array. For instance to calculate the value for $\text{SUM}(A[0, 0] : A[7, 4])$ we have to add *OL*$[6, 3]$, *OL*$[7, 3]$, *OL*$[6, 4]$ and *RP*$[7, 4]$, resulting in 4 accesses and returning 142.

In general, a query for $\text{SUM}(A[0, \ldots, 0] : A[z_1, \ldots, z_d])$, i.e., a query that sums the values of all array cells up to $Z = (z_1, \ldots, z_d)$, accesses exactly one value in *RP* and some values in the overlay box B_Z that covers Z. Let B_Z be anchored at $(b_1, \ldots, b_d)$. Then all overlay cells in B_Z that together aggregate the range $A[0, \ldots, 0] : A[z_1, \ldots, z_d]$, excluding the cells covered by overlay box B_Z, must be accessed. These are all cells $X = (x_1, \ldots, x_d)$ that satisfy for all dimensions i: $x_i = b_i$ or $x_i = z_i$, excluding cell $(z_1, \ldots, z_d)$ which is not a used overlay cell. Intuitively all overlay cells have to be accessed whose coordinates can be obtained from Z by replacing one or more of the z_i with the corresponding b_i. In total $2^d - 1$ overlay cells must be accessed. The overall cost for a query anchored at $(0, \ldots, 0)$ therefore sums to 2^d. Since an arbitrary range sum query can be computed by adding/subtracting the results of at most 2^d of these queries, the overall worst case query cost

OL	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		

RP	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 4: Overlay *OL* and array *RP* with overlay boxes drawn for reference (computed for array A)

OL	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	*		17			42		
6	21	21	31	88	20	51	181	20	40
7	0			8			14		
8	0			20			32		

RP	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	18	23	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 5: Effects of an update to cell $(1, 5)$ (marked with a star)

is 4^d. For a certain data cube its dimensionality d is fixed, i.e., we obtain a worst query cost that is constant and independent of the size of the query range.

Determining the update cost requires a more detailed analysis. We first illustrate the update process on our two-dimensional example array (Figure 5). Let $A[1, 5]$ be updated with the value 5 (replacing the former value 3) and let B be the overlay box anchored at $(0, 3)$, i.e., the box that covers $(1, 5)$. In *RP* only cells covered by B that reside to the lower right of $(1, 5)$ including the cell itself, have to be updated (shaded area in *RP*). In *OL*, for overlay boxes in the same row as B to its right, the corresponding column values are affected ($(3, 5)$ and $(6, 5)$), similar for the overlay boxes in the same column as B below it. Finally the anchor cells $(3, 6)$ and $(6, 6)$ have to be updated as well. In Figure 5 the affected overlay cells are shaded.

To keep the description of the general case simple, we assume that k, the side-length of an overlay box, evenly divides n, the side-length of the data cube. Let $Z = (z_1, \ldots, z_d)$ be the updated cell and B_Z be the overlay box anchored at $(b_1, \ldots, b_d)$ that covers Z. The cost of an update to Z is the sum of the cost of updating the overlay cells and the cost of updating *RP*. Regarding the relative prefix sum array *RP*, only cells inside overlay box B_Z can be affected by the update. To be more precise, only those cells in B_Z whose indexes are each at least as great as the corresponding index of Z are to be updated, resulting in a worst case cost of k^d.

Next we describe which overlay cells need to be updated. In general, these are all overlay cells that include Z in their aggregation. Let for each dimension i: $S_i = \{b_i+k; b_i+2k; b_i+3k; \ldots; n-k\}$. Intuitively S_i contains the coordinates of the anchors of the affected overlay boxes (the boxes to the "lower right" of B_Z in two-dimensional terminology) in the i-th dimension. Let for each dimension i: $T_i = \{z_i; z_i+1; z_i+2; \ldots; b_i+(k-1)\}$. This set intuitively contains the coordinates of the affected overlay cells inside a certain overlay box (the cells to the "lower right" of Z in two-dimensional terminology) in the i-th dimension. Then the overlay cells $O = (o_1, \ldots, o_d)$ that need to be updated are those that satisfy

$$\forall i : o_i \in \begin{cases} S_i & \text{, if } z_i = b_i \\ S_i \cup T_i & \text{, otherwise} \end{cases}$$

and do not belong to overlay box B_Z. In the example of Figure 5 the sets become $S_x = \{3; 6\}$, $T_x = \{1; 2\}$,

$S_y = \{6\}$ and $T_y = \{5\}$. Since $z_x = 1 > 0 = b_x$ and $z_y = 5 > 3 = b_y$ the affected overlay cells are all cells (o_x, o_y) where $o_x \in \{1; 2; 3; 6\}$ and $o_y \in \{5; 6\}$, minus $(1, 5)$ and $(2, 5)$ which fall into the same overlay box as $(1, 5)$.

Obviously the greatest number of affected overlay cells results from choosing $z_i \neq b_i$ and z_i as small as possible[2]. S_i has at most $n/k - 1$ elements, T_i can have up to $k - 1$ elements[3]. Since there are at most $(n/k - 1 + k - 1) = (n/k + k - 2)$ possible values for each dimension, there are at most $(n/k + k - 2)^d$ overlay cells that satisfy the above formula. Among those, $(k - 1)^d$ cells fall into overlay box B_Z[4]. Altogether $(n/k + k - 2)^d - (k - 1)^d$ overlay cells must be updated in the worst case. Note, that this bound is tight, since it is met for an update on $(1, \ldots, 1)$. For $n/k \geq 2$ the cost of updating the overlay cells determines the worst overall update cost. Therefore the worst update cost is obtained when cell $(1, \ldots, 1)$ is updated, resulting in a cost of $(n/k + k - 2)^d - (k - 1)^d + (k - 1)^d = (n/k + k - 2)^d$. This value is minimized for $k = \sqrt{n}$. The worst update cost therefore is $O(n^{d/2})$ (compare to $O(n^d)$ for the PS technique).

3.2 The Dynamic Data Cube (DDC)

The Dynamic Data Cube [5] provides sub-linear performance ($O(\log^d n)$) for both range sum queries and updates on the data cube. The method supports dynamic growth of the data cube in *any* direction and gracefully manages clustered data and data cubes. The DDC method utilizes a tree structure which recursively partitions array A into a variant of overlay boxes. Each overlay box will contain information regarding relative sums of regions of A. By descending the tree and adding these sums, we will efficiently construct sums of regions which begin at $A[0, \ldots, 0]$ and end at any arbitrary cell in A. To calculate complete region sums from the tree, we again make use of the inverse property of addition as illustrated in Figure 2. We will first describe the overlay box variant, then describe their use in constructing the Dynamic Data Cube.

3.2.1 Overlay Variant

For the DDC we define an overlay as before, i.e., as a set of disjoint hyper-rectangles (hereafter called "boxes") of equal size that completely partition the set of cells of array A into non-overlapping regions. However, these overlay boxes differ from those in RPS in the values they store and in the number of overlay boxes used to partition the data space. Referring to Figure 6, S is the subtotal cell, while x_1, x_2, x_3 are row sum cells in the first dimension and y_1, y_2, y_3 are row sum cells in the second dimension. Each box stores exactly $k^d - (k - 1)^d$ values; the other cells covered by the overlay box are not needed in the overlay, and would not be stored. Values stored in an overlay box provide sums of regions *within* the overlay box. Figure 6 demonstrates the calculation of those values. The row sum values shown in the figure are equal to the sum of the associated shaded cells in array A. Note that row sum values are cumulative; i.e., y_2 includes the value of y_1, etc. Formally, given an overlay box anchored at $A[i_1, i_2, \ldots, i_d]$, the row sum value contained in cell $(i_1, i_2, \ldots, j, \ldots, i_d)$ is equal to $\mathrm{SUM}(A[i_1, i_2, \ldots, i_d] : A[i_1, i_2, \ldots, j, \ldots, i_d])$.

3.2.2 Constructing the Dynamic Data Cube

Overlay boxes are used in conjunction with a tree that recursively partitions array A. We now describe its construction (Figure 7). The root node of the tree encompasses the complete range of array A. It forms children by dividing its range in each dimension in half. It stores a separate overlay box for each child. Each of its children are in turn subdivided into children, for which overlay boxes are stored. This recursive partitioning continues until the leaf level. Thus, each level of the tree has its own value for the overlay box size k; k is $n/2$ at the root

[2]The choice of z_i determines the value of b_i. The smaller z_i, the smaller b_i, i.e., the more elements in S_i.
[3]This is because $z_i \neq b_i$, i.e., $z_i \geq b_i + 1$.
[4]These are all cells where $\forall i : o_i \in T_i$.

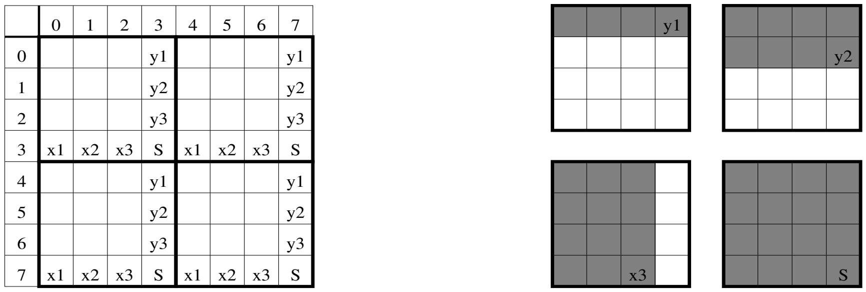

Figure 6: Partitioning of array A into overlay boxes and calculation of overlay values

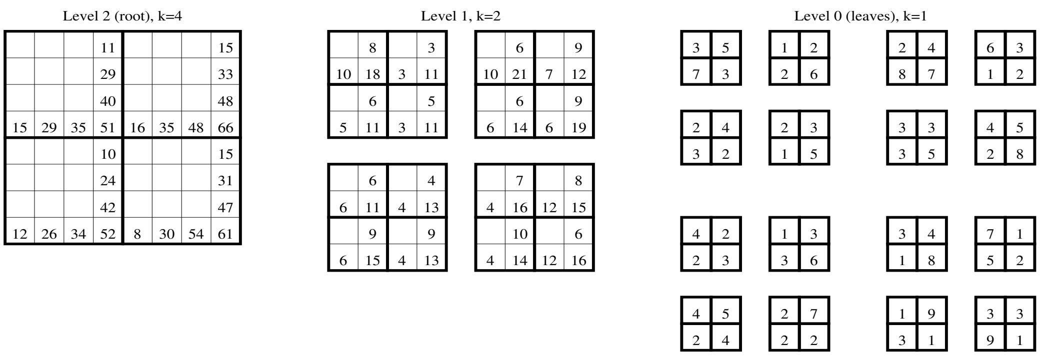

Figure 7: Dynamic Data Cube (computed for the data in array $A[0,0] : A[7,7]$)

of the tree, and is successively divided in half for each subsequent tree level. We define the leaf level as the level wherein $k = 1$. When $k = 1$, each overlay box contains a single cell; since a single-cell overlay box contains only the subtotal cell, the leaf level contains the values stored in the original array A.

Overlay box values are stored in special structures to guarantee the sublinear query and update times. For two-dimensional overlays (i.e., $d = 2$) we do not store the values of an overlay box in arrays. Instead a hierarchical structure is used (B ^{c}-tree, see [5]) that has an access and update cost of $O(\log n)$. For higher dimensional data cubes ($d > 2$) we make the observation that the surfaces containing the overlay values of a d-dimensional overlay box are $(d-1)$-dimensional. Thus, the overlay box values of a d dimensional data cube can be stored as $(d-1)$-dimensional data cubes using Dynamic Data Cubes, recursively. The recursion stops for $d = 2$.

3.2.3 Query and Update Operations

The range sum for any query anchored at $(0, \ldots, 0)$ is obtained by only accessing overlay values. We describe this process for a query $\mathrm{SUM}(A[0, \ldots, 0] : A[z_1, \ldots, z_d])$, i.e., a query that sums the values of all array cells up to $Z = (z_1, \ldots, z_d)$. The query process begins at the root of the tree. The algorithm checks the relationship of cell Z and the overlay boxes in the node. When an overlay box covers Z, a recursive call to the function is performed, using the child associated with the overlay box as the node parameter (i.e., the algorithm descends the tree for that case). When Z comes before the overlay box in any dimension (i.e., has a smaller index than each cell covered by the overlay box in that dimension), the query region does not intersect the overlay box, and therefore this box does not contribute a value for the sum. When Z comes after the overlay box in every dimension (i.e., has a greater index than each cell covered by the overlay box in every dimension), the query includes the entire overlay box, and the box contributes the subtotal to the sum. Otherwise the cell is neither before nor after the box, i.e., the query area intersects the overlay box, and the box contributes the corresponding overlay value (a row sum value in two-dimensional terminology) to the sum.

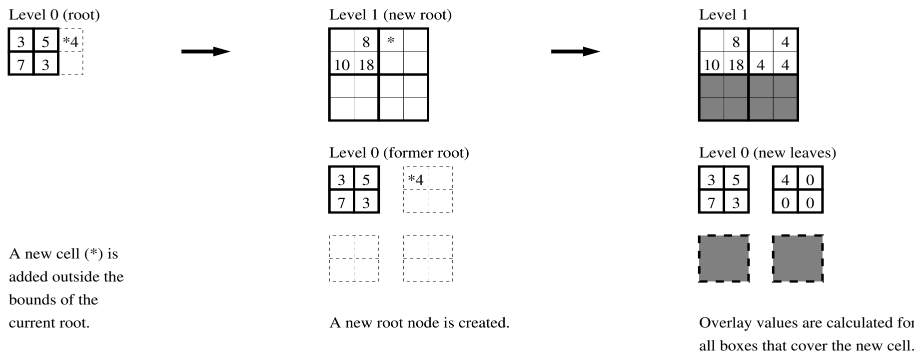

Figure 8: Example for the growth of the Dynamic Data Cube (shaded areas do not store values)

Since overlay boxes at the same tree level are non-intersecting, at most one child will be descended at a tree level. The contribution of overlay boxes that intersect the query area but do not cover Z is obtained by accessing a single overlay value. There are at most $2^d - 1$ overlay boxes in a node that can have this property. In sum at most $(2^d - 1)\log_2 n = O(\log n)$ overlay values are accessed. Due to the recursive way of storing the overlays, accessing a single overlay value costs $O(\log^{d-1} n)$, resulting in an overall query cost of $O(\log^d n)$ (for details see [5]).

To perform an update on cell Z the DDC tree has to be descended in a way similar to the query process. Even though the cells of an overlay box store cumulative values, the balanced query/update cost of the B^c-trees (for $d = 2$) together with the recursive way of storing higher dimensional boxes result in a worst case update cost of $O(\log^d n)$ (for details see [5]).

3.2.4 Dynamic growth of the cube

Neither the Prefix Sum (PS), nor the Relative Prefix Sum (RPS), nor the Hierarchical Cubes (HC) [2] methods address the growth of the data cube. Instead, they assume that the size of each dimension is known a priori. For some applications, however, it is more convenient and space efficient to grow the size of the data cube dynamically to suit the (size of the) data. For instance, an attribute might have a large domain, but the data cube only contains non-empty cells that can be addressed by a much smaller range of values for that attribute. Take for example astronomical databases where new telescopes allow discovering stars in greater distance; or commercial applications where new products and customers are added or deleted from time to time.

The PS, RPS and HC methods would store each single cell in non-populated areas, wasting a huge amount of space. The Dynamic Data Cube, on the other hand, could start by building the smallest data cube that contains all non-empty cells. As soon as a cell is inserted that lies outside the current data cube, the data cube "grows" into the required direction. New roots are created successively, each time doubling the size of the data cube in each dimension, until the new root encompasses the new cell. This update process is incremental, i.e., the old tree structure appears unchanged as the descendent of the new root (see Figure 8 for a simple example). Only one overlay box at each tree level is affected by an update; therefore, we will create only one child node and overlay box per tree level during this process.

This incremental construction of the Dynamic Data Cube is naturally suited to clustered data and data that contains large, non-populated regions. Where data does not exist, overlay boxes will not be instantiated; thus, the Dynamic Data Cube avoids the storage of empty regions. Since overlay boxes are self-contained, there is no cascading update problem associated with adding a new cell. The Dynamic Data Cube allows graceful growth of the data cube in any direction, making it more suitable for applications which involve change or growth. Note that the PS and RPS techniques could be augmented by methods to handle a data cube growth by appending rows. Handling growth in *any* direction, however, will be very costly.

4 Conclusion

In the near future an increasing number of applications will require or be enabled by providing fast and frequent updates on data cubes and avoiding long down-times of the data analysis tools. Together, the RPS and DDC methods offer a range of options for implementing data cubes in such dynamic environments. The Dynamic Data Cube provides balanced sub-linear performance for queries and updates. It is suitable for dynamic environments where queries and updates are both frequent; where data cubes are very large; where data is clustered and sparse; and where the data can grow in any direction relative to the original data (i.e., updates are not append-only). The Relative Prefix Sum technique does not offer this flexibility, but has its merits for applications that do not deal with frequent updates but require fast answers within a guaranteed time limit.

The obvious disadvantage of our methods compared to PS and HC is the usage of additional space. While PS and HC require exactly the same amount of storage as the original data cube ($O(n^d)$), RPS and DDC need to store the overlay values. In the case of RPS the storage overhead is provably within small bounds. Since each overlay box of size k^d stores $k^d-(k-1)^d$ values, the ratio of RPS's total storage requirements to the requirements of the original data cube is $2-((k-1)/k)^d$, i.e., less than 2. For $d=4$ and $k=100$ RPS uses only 4% more storage. In the case of DDC it is obvious that the lowest tree levels consume the most storage. By deleting a certain number of those levels, one can trade off query speed for storage space, or conversely bring the DDC within delta of the storage required by the Prefix Sum method (for details see [5]). Also, in the case of DDC the worst case storage overhead will only occur for dense data cubes. For empty chunks whole subtrees will not be created, saving a considerable amount of space for practical applications.

We are currently developing new techniques that apply the basic ideas of the presented approaches to high-dimensional and sparse data sets.

References

[1] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *Proc. 13th ICDE*, 1997.

[2] C.-Y. Chan and Y. E. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. 25th VLDB*, 1999.

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

[5] S. Geffner, D. Agrawal, and A. El Abbadi. The dynamic data cube. In *Proc. EDBT*, 2000. To appear.

[6] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *Proc. 15th ICDE*, 1999.

[7] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, pages 29–53, 1997.

[8] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD*, 1997.