The Dynamic Data Cube

S. Geffner D. Agrawal A. El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106
{sgeffner, agrawal, amr}@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, AVERAGE) over all selected cells in a data cube, where the selection is specified by providing ranges of values for numeric dimensions. We present the Dynamic Data Cube, a new approach to range sum queries which provides efficient performance for both queries and updates, which handles clustered and sparse data gracefully, and which allows for the dynamic expansion of the data cube in any direction.*

1 Introduction

The data cube [GBLP96], also known in the OLAP community as the multidimensional database [OLA96][AGS97], 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 of sales information maintained by a company. One may construct a data cube from the database with SALES as a measure attribute and CUSTOMER_AGE and DATE_AND_TIME as dimensions. Such a data cube provides aggregated sales information for the enterprise over time; for example, *what were the total sales to 45-year-old customers on the 8th of December?* Range sum queries are useful analysis tools when applied to data cubes. A range sum query applies an aggregate operation (e.g., SUM, AVERAGE) to the measure attribute within the range of the query. An example is *find the average daily sales to customers between the ages of 27 and 45 during the time period December 7 to December 31*. 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) [Cod93]. Since the introduction of the data cube, there has been considerable research in the database community regarding the computation of data cubes [AAD+96], for choosing subsets of the data cube to precompute [HRU96] [GHRU97], for constructing estimates of the size of multidimensional aggregates [SDNR96] and for indexing precomputed summaries [SR96] [JS96]. Approximating the data cube using wavelets has also been examined [VWI98].

Ho, Agrawal, Megiddo and Srikant [HAMS97] have presented an elegant algorithm for computing range sum 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 hampered by its update cost, which in the worst case requires recalculating all the entries in an array of the same size as the entire data cube. In a recent paper [GAES99], we have presented an algorithm for computing range sum queries in data cubes which we call the *relative prefix sum* approach. The relative prefix sum method improves upon earlier results in that it prevents unconstrained cascading updates, and consequently achieves a reduction in update complexity while maintaining constant time queries. Nevertheless, it still incurs substantial update costs in the worst case, on the order of the square root of the size of the cube.

These methods, which each provide constant-time query performance, are suitable for applications in which the underlying data is dense and update performance is irrelevant. In current data analysis applications, update complexity is rarely considered to be of significant importance. Most analysis systems are oriented towards batch updates, and for a wide variety of current-day business applications this is considered sufficient. Yet, the batch updating paradigm, a holdover from the 1960's computing environment, is tremendously limiting to the field. In the technological sector, we are all familiar with the notion of an *enabling threshold*, the point at which new applications become efficient enough to be practical. As an example, before 1990, Internet use was essentially limited to universities, and few applications existed. With the arrival of WWW and HTML, along with the search engines they made possible, an enabling threshold had been crossed; the Internet became efficient and easy enough to be adopted by the public en masse, and many new applications arose.

One of the goals of research is to find and cross enabling thresholds. Again, in current practice, data cubes are used almost exclusively by data analysts, using relatively expensive systems that first batch load data, then permit read-only querying. This model of interaction clearly limits what is possible. The prefix sum method is a good example of present-day cutting-edge data cube technology. During updates, it requires 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 applications. What if the size of the data cube were a terabyte? What if batch updates occur every minute (think Internet commerce)? Table 1 compares update costs for various methods of computing range sum queries in data cubes when the number of dimensions is 8. In the table, n is the size of each dimension, while d is the number of dimensions; thus, the size of the complete data cube is n^d. Even in the relatively smaller data cubes, the performance difference is striking. When $n=10^2$, the size of each dimension is only 100 elements; yet, with d=8, the full data cube is 10^{16} cells. To handle a single update at this data cube size, the prefix sum method requires on the order of 10^{10} times more instructions than the Dynamic Data Cube. On a hypothetical 500MIPS processor, excluding I/O and other costs and ignoring constants in the formulas, the prefix sum method may require more than 6 months of processing to update a single cell in the data cube; in such a case, even batch updating is not practical. The Dynamic Data Cube can update that same cell in under 0.008 seconds. The relative prefix sum approach also quickly becomes impractical. When $n=10^4$, the relative prefix sum method requires 231 days to update a single cell in the data cube, whereas the Dynamic Data Cube requires under 2 seconds. Figure 1 presents the update functions in graphical form for a range of data cube sizes.

How many potential applications of data cube techniques are currently infeasible due to the high cost of updates? Astronomers, for example, might wish to do some data analysis on the billions of stars they are discovering. Stock brokers might wish to dynamically analyze the implications of millions of trades as they occur. Business leaders might wish to construct interactive "what-if" scenarios using their data cubes, in much the same way that they construct "what-if" scenarios using spreadsheets now. The fact that there are significant impediments to dynamic updates in the data cube prevents many potential applications from being even considered, since those applications are clearly not practical at this time. As an industry, we need to fundamentally reduce the barriers to dynamic updates in very large data cubes so that new and interesting applications become possible.

In addition, for many application domains data is sparse or clustered. Examples include most geographically-based information, such as geographically oriented business data (e.g. sales by region, median income of households by region, etc.), and scientific measurements (e.g., levels of carbon monoxide production at numerous points on the Earth's surface, locations of stars in space). Still other applications require that data be allowed to grow dynamically in any direction, rather than in a single dimension as with append-only databases. Current techniques do not handle these cases well. For these and other potential application domains, we desire a method that achieves sublinear performance for both queries and updates. The method should permit the data cube to grow dynamically in any direction to suit the underlying data, and should handle sparse or clustered data efficiently.

Table 1. Update cost functions by method, d=8. Values are rounded to the nearest power of 10.

Update Cost Functions by Method, d=8				
n	Full Data Cube Size $=n^d$	Prefix Sum $=n^d$	Relative PS $=n^{d/2}$	Dynamic Data Cube $=(\log_2 n)^d$
10	10^8	10^8	10^4	10^4
10^2	10^{16}	10^{16}	10^8	10^6
10^3	10^{24}	10^{24}	10^{12}	10^7
10^4	10^{32}	10^{32}	10^{16}	10^9

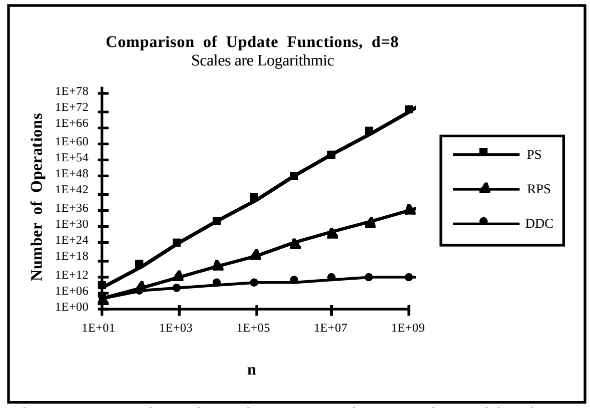

Figure 1. Update functions. Scales are logarithmic.

Contribution We present the Dynamic Data Cube, a method that provides sublinear performance for both range sum queries and updates on the data cube. The method supports dynamic growth of the data cube in any direction; it gracefully manages clustered data and data cubes that contain large regions of empty space.

Paper Organization The remainder of the paper is organized as follows. In Section 2, we present the model of the range sum problem, and discuss several previous approaches. In Section 3, we introduce the *Basic Dynamic Data Cube* as a foundation for later sections. We present basic query and update methods. We present a performance analysis of the Basic Dynamic Data Cube, concluding that the method still has considerable update complexity as the dimensionality of the data cube increases. In Section 4, we present the *Dynamic Data Cube* and analyze the performance characteristics of the method. We show that it achieves sublinear queries and updates, and we discuss various performance considerations. Section 5 addresses dynamic growth of the data cube. We demonstrate that the Dynamic Data Cube is more suited than previous methods to dynamic growth of the cube, and that it handles clustered data more efficiently. Section 6 concludes the paper.

2 Problem Statement and Previous Solutions

Assume the data cube has one measure attribute and d feature attributes (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_AND_TIME. Each dimension has a size n_i, which represents the number of distinct values in the dimension. Initially we assume that this size is known a priori; in Section 5 we will expand our analysis to dynamic environments. 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 2, 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 formulae more concisely. Thus, let the size of each dimension be n, i.e. $n=n_1=n_2=...=n_d$. 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.

Array A

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

Figure 2. The data cube represented as an array A.

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_AND_TIME, the cell at A[37, 220] contains the total sales to 37-year-old customers on day 220. 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 220 to 222 would be answered by summing the cells A[37, 220], A[37, 221], and A[37, 222]. 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.

The *prefix sum* approach [HAMS97] achieves O(1) complexity for queries and $O(n^d)$ complexity for updates. 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. Figure 3 shows the array P employed by the prefix sum approach. Each cell P[i,j] in array P stores the sum of all cells that precede it in array A, i.e., SUM(A[0,0]:A[i,j]). Using the prefix sum method, arbitrary range sum queries can be evaluated by adding and subtracting a constant number of cells in array P. In recent work, we presented the *relative prefix sum* approach [GAES99]. The relative prefix sum approach achieves O(1) complexity for queries and $O(n^{d/2})$ for updates.

Array P

Index	0	1	2	3	4	5	6	7
0	3	8	9	11	13	17	23	26
1	10	18	21	29	39	50	57	62
2	12	24	29	40	53	67	78	88
3	15	29	35	51	67	86	99	117
4	19	35	42	61	80	103	123	142
5	21	40	50	75	95	126	151	172
6	25	49	61	93	114	154	182	206
7	27	55	69	103	127	168	205	230

Figure 3. Array P used in the prefix sum method.

Both the prefix sum approach and the relative prefix sum approach make use of a property of range sums in data cubes that is a consequence of the inverse property of addition. Figure 4 presents the essential idea: 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. This technique requires a constant number of region sums that is related to the number of dimensions. We note that all such regions begin at cell A[0,0] and extend to some other cell in A. In the prefix sum method, the array P stores these region sums directly, and uses them to answer arbitrary queries as illustrated in Figure 4. In the relative prefix sum method, these sums are stored indirectly in a manner which improves update complexity.

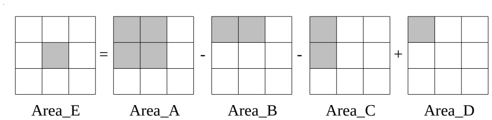

Figure 4. A geometric illustration of the two dimensional case: Sum(Area_E) = Sum(Area_A) - Sum(Area_B) - Sum(Area_c) + Sum(Area_D).

While these methods provide constant time queries, in the worst case they incur update costs proportional to the entire data space. This update cost results from the very dependencies in the data which allow these methods 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. Figure 5 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 P 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, this update complexity is $O(n^d)$. The relative prefix sum method constrains cascading updates somewhat, but is still subject to this effect.

Array P

Index	0	1	2	3	4	5	6	7
0	3	8	9	11	13	17	23	26
1	10	18 *	21	29	39	50	57	62
2	12	24	29	40	53	67	78	88
3	15	29	35	51	67	86	99	117
4	19	35	42	61	80	103	123	142
5	21	40	50	75	95	126	151	172
6	25	49	61	93	114	154	182	206
7	27	55	69	103	127	168	205	230

Figure 5. Array P update example.

Index	0	1	2	3	4	5	6	7
0				Y_1				Y_1
1				Y_2				Y_2
2				Y_3				Y_3
3	X_1	X_2	X_3	S	X_1	X_2	X_3	S
4				Y_1				Y_1
5				Y_2				Y_2
6				Y_3				Y_3
7	X_1	X_2	X_3	S	X_1	X_2	X_3	S

Figure 6. Partitioning array A into overlay boxes.

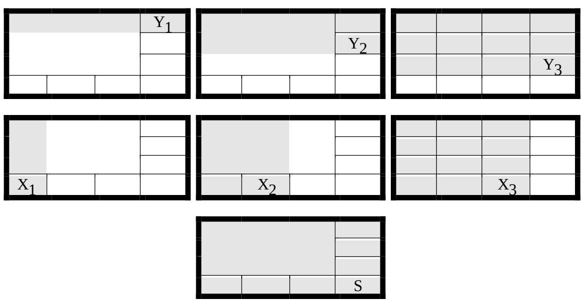

Figure 7. Calculation of row sum values.

Index	0	1	2	3	4	5	6	7
0				11				15
1				29				33
2				40				48
3	15	29	35	51	16	35	48	66
4				10				15
5				24				31
6				42				47
7	12	26	34	52	8	30	54	61

Figure 8. Array A partitioned into overlay boxes.

3 The Basic Dynamic Data Cube

In this section, we describe the Basic Dynamic Data Cube as a foundation for later sections. The method utilizes a tree structure which recursively partitions array A into *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[0,0] and end at any arbitrary cell in A. To calculate complete region sums from the tree, we also make use of the inverse property of addition as illustrated in Figure 4. We will first describe overlays, then describe their use in constructing the Basic Dynamic Data Cube. As motivation to Section 4, we will analyze the performance of the basic tree, and show that its update complexity is still problematic.

3.1 Overlays

We define an *overlay* as a set of disjoint hyperrectangles (hereafter called "boxes") of equal size that completely partition cells of array A into non-overlapping regions. For simplicity in presentation, we will assume that the size of A in each dimension is 2^i for some integer i. We also define several terms for use later in the paper. We denote the length of the overlay box in each dimension as k. 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 $\forall i((a_i \le x_i) \wedge (a_i + k > x_i))$.

Figure 6 shows array A partitioned into overlay boxes. Each dimension is subdivided in half; in this two-dimensional example, there are four resulting boxes. In the figure, k=4; i.e., each box in the figure is of size 4×4. The boxes are anchored at cells (0,0), (0,4), (4,0), and (4,4). Each overlay box corresponds to an area of array A of size k^d cells; thus, in this example each overlay box covers 4^2 = 16 cells of array A.

Each overlay box stores certain values. 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. Row sum values provide the cumulative sums of rows, in each dimension, of cells covered by the overlay box. Figure 7 demonstrates the calculation of row sum values; the row sum values shown in the figure are equal to the sum of the associated shaded cells in array A. Row sum value Y_1 is the sum of all cells within the overlay box in the row containing cell Y_1. Row sum value Y_2 is the sum of all cells within the overlay box in the row containing cell Y_2, plus Y_1. Row sum value X_1 is the sum of all cells within the overlay box in the column containing cell X_1. Row sum value X_2 is the sum of all cells within the overlay box in the column containing X_2, plus X_1. Note that row sum values are cumulative; i.e., X_2 includes the value of X_1, and X_n includes the values of $X_1..X_{n-1}$. Formally, given an overlay box anchored at $A[i_1, i_2, ..., i_d]$, the row sum value contained in cell $[i_1, i_2, ..., j, ..., i_d]$ is equal to $SUM(A[i_1, i_2, ..., i_d]:A[i_1, i_2, ..., j, ..., i_d])$. The subtotal value S is the sum of all cells in A covered by the overlay box. Formally, an overlay box anchored at $A[i_1, i_2, ..., i_d]$ has a subtotal value that is equal to $SUM(A[i_1, i_2, ..., i_d]:A[i_1+k-1, i_2+k-1, ..., i_d+k-1])$.

Figure 8 shows array A partitioned into overlay boxes of size 4×4. The subtotal in cell [3,3] is equal to the sum of all cells from A covered by the first overlay box, i.e. Sum(A[0,0] .. A[3,3]) = 51. The row sum in overlay cell [0,3] = A[0,0] + A[0,1] + A[0,2] + A[0,3] = 3+5+1+2 =

11. The row sum in overlay cell [1,3] = A[0,0] + A[0,1] + A[0,2] + A[0,3] + A[1,0] + A[1,1] + A[1,2] + A[1,3] = 3+5+1+2+7+3+2+6 = 29. Similarly, the row sum in overlay cell [3,0] = A[0,0] + A[1,0] + A[2,0] + A[3,0] = 3+7+2+3 = 15.

3.2 Constructing the Basic Dynamic Data Cube

We now describe the construction of the Basic Dynamic Data Cube, which organizes overlay boxes into a tree to recursively partition array A (Figure 9). The root node of the tree encompasses the complete range of array A. The root node 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 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.

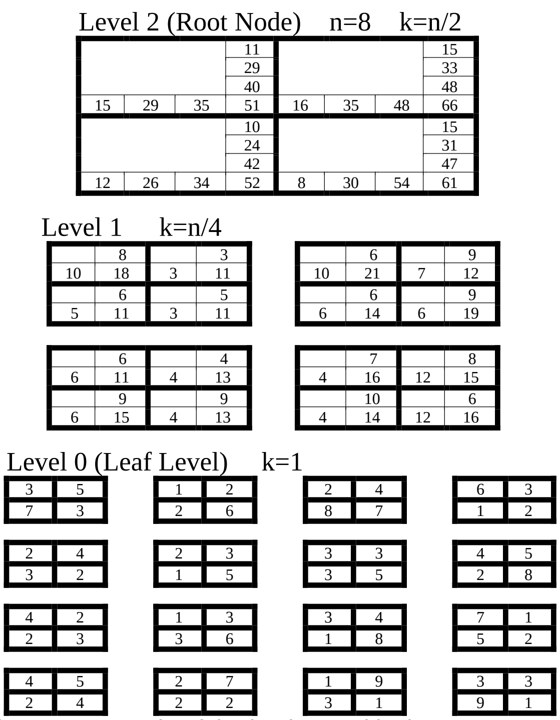

Figure 9. Example of the basic tree (d=2).

3.2.1 Queries

The Basic Dynamic Data Cube can be used to generate the sum of any region of A which begins at A[0,0] and ends at an arbitrary cell c in A; we will refer to such a region as the *target region*, and to c as the *target cell*. Figure 10 presents a range sum query algorithm. The query process begins at the root of the tree. Using the target cell, the algorithm checks the relationship between the target cell and the overlay boxes in the node. When an overlay box covers the target cell, a recursive call to the function is performed, using the child associated with the overlay box as the node parameter. When the target cell comes before the overlay box in any dimension, the target region does not intersect the overlay box, and the box contributes no value to the sum. When the target cell is after the overlay box in every dimension, the target region includes the entire overlay box, and the box contributes its subtotal cell to the sum. When the cell is neither before nor completely after the overlay box, the target region intersects the overlay box, and the box contributes a row sum value to the sum.

Exactly one child will be descended at each level of the tree. This property follows from the construction of overlays. Overlay boxes completely partition array A into disjoint regions. Therefore, the target cell must fall within only one overlay box at a given level of the tree. Given a node and its overlay boxes, the target cell will fall within one box, and outside the others. Consider the boxes that do not enclose the target cell. Overlay boxes store the cumulative sums of rows in the region covered by the overlay box. Therefore, the contribution of these regions can be determined directly from the overlay box values; no descent is necessary. When the target cell falls within an overlay box, we must descend to the child associated with that overlay box. Therefore, exactly one child will be descended in the tree at any given node, and queries are of complexity O(log n).

```
/* Function CalculateRegionSum
   Returns the contribution of node h and its subtree to
   the sum of the region A[0,...,0]:cell */
int CalculateRegionSum (DDCTreeNode h, Cell cell) {
   int sum=0;  /* running total of sum contributed by this
                  node and its subtrees */
   int i;      /* index variable */

   //naive code for clarity's sake --
   //  production code would only check overlay boxes
   //  which intersect the target region
   for (i=0; i<NUM_OVERLAY_BOXES_PER_NODE; i++) {
      //   ** NUM_OVERLAY_BOXES_PER_NODE = 2^d  **
      if (Covers(h.box[i], cell) {
         if (h is a leaf) sum+=h.box[i].subtotal;
         else sum+=CalculateRegionSum(h.child[i], cell);
      } else {
         if (CellBeforeBox(h.box[i], cell)) sum+=0;
         else if (CellCompletelyAfterBox(h.box[i]))
             sum+=box.subtotal;
         else
             sum+=the appropriate row sum value
                  from this overlay box;
      }
   }

   return sum;
}
```

Figure 10. Query algorithm, Basic Dynamic Data Cube.

An example of the query process is presented in Figure 11. We will calculate the region sum of the region that begins at A[0,0] and ends at cell * in the figure. For illustrative purposes only, we have labeled the overlay boxes for the four children of the root **Q**, **R**, **S** and **T**. Each of these overlay boxes contributes at most one value to the sum of the target region. Overlay box **Q** contributes its subtotal (51), since the target region includes all of the area covered by **Q**. **R** contributes its row sum value (48), which represents the sum of all the rows in **R** that are contained in the target region. Likewise, **S** contributes its row sum value (24). By summing these three values, the sum of all cells of A within the shaded region of the root node is obtained. The target cell lies within **T**, so we must descend to the child associated with **T** to calculate the remaining sum of the target region. Descending to tree level one, we have labeled the overlay boxes of the appropriate node **U**, **V**, **W** and **Z**. **U** contributes its subtotal cell (16). Note that not all overlay boxes in a node always contribute to the sum. In this case, **W** and **Z** do not contribute any values to the sum of the target region, since they do not intersect it. Since the target cell falls within **V**, we must descend to the child associated with **V**. At the leaf level, we have labeled the overlay boxes of the appropriate leaf node **L**, **M**, **N** and **O**. Note that each overlay box at the leaf level contains only its the subtotal cell. **L** contributes its subtotal cell (7), and **N** contributes its subtotal cell (5), while **M** and **O** do not contribute to the target region sum. The total region sum thus consists of 51+48+24+16+7+5=151, which is the sum of all cells in array A in the range A[0,0] to the target cell A[6,6] (Figure 11a).

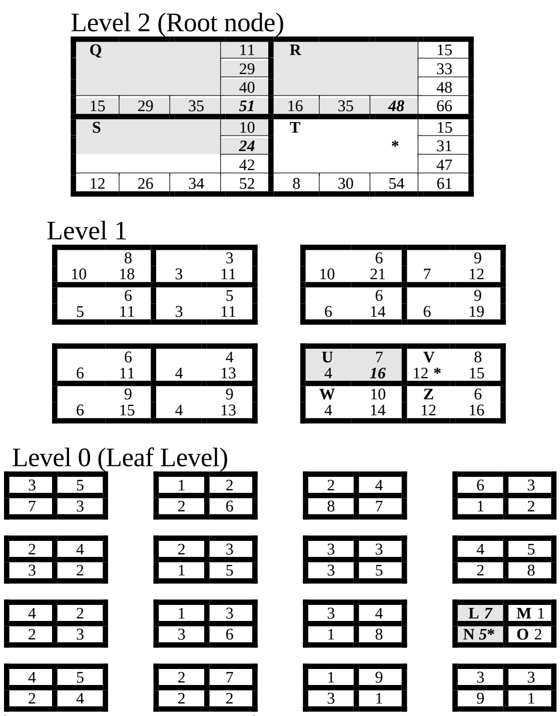

Figure 11. Query example.

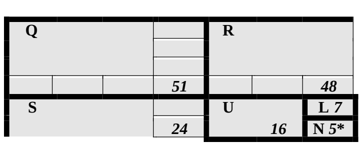

Figure 11a. Individual components of the range sum.

3.2.2 Updates The value of a cell can be updated by descending a single path in the tree. This follows from the construction of overlay boxes. At any level of the tree, an update to a cell effects only the overlay box that contains it; other overlay boxes are unaffected. The update algorithm (Figure 12) makes use of a bottom-up approach. It first traverses the tree to the leaf associated with the target cell. When the leaf is reached, the algorithm determines the difference between the old and new values of the cell, and stores the new value into the cell. The difference value is used to update overlay box values in ancestor nodes of the tree.

```
/* Function UpdateCell
   Updates the data cube in response to a change in the
   value of a cell in array A. Returns the difference
   between the old and new values of cell. */
int UpdateCell(DDCTreeNode h, Cell cell, int newValue) {
  int oldValue;
  int difference;
  int i;

  i = the index of the overlay box in h that covers cell;

  if (h is not a leaf) {
    difference=UpdateCell(h.child[i], cell, newValue);
      /*  difference=oldValue-newValue  */
    offset[] = offset of cell within h.box[i]
               in each dimension;
    for each set of row sum values { /* d sets */
      add difference to all row sum values ≥ offset in
      that dimension
    }
    return difference;
  } else { /* h is a leaf */
    oldValue=h.box[i].subtotal;
    h.box[i].subtotal=newValue;
    return (oldValue-newValue); /* return difference */
  }
}
```

Figure 12. Update algorithm, Basic Dynamic Data Cube.

Referring to Figure 11, we will assume that the value of cell ***** is to be updated from 5 to 6. The update function will recursively call itself, traversing down the tree to the leaf containing *****; it begins at the root and descends the child associated with overlay box **T**, then of **V**, then reaches the leaf. The cell to be updated is associated with overlay box **N**. The difference between the old and new values of the cell is +1. The algorithm stores the new value, 6, into **N**. The difference is returned up the calling chain, where the values in overlay box **V** will be updated. All values to the right or below the changed cell must be updated; in this case, the row sum value (12) and the subtotal cell (15) must be increased by `difference`. The recursion unwinds to the root level, where the values in overlay box **T** are updated. The row sum values (31), (47), and (54), and the subtotal cell (61), must be increased by `difference`.

Only one overlay box is updated at each tree level; therefore, the cost of updating the Basic Dynamic Data Cube is O(log n) plus the cost of updating the values in these overlay boxes. However, updates to overlay boxes can be expensive. Assume an array A of two dimensions, where the size of each dimension is n. Further assume a tree on array A as we have described. As noted earlier, each overlay box contains $(k^d - (k-1)^d)$ values; however, for the two dimensional case we can observe from the figures that the number of row sum values, not including the subtotal cell, is equal to d(k-1). Thus, at the root level of the tree, each overlay box must store 2(n/2 -1) row sum cells, or O(n) cells. Row sum values are cumulative sums of rows. In the worst case, updating a single cell covered by an overlay box may require that every row sum value in the overlay box be updated; thus, updating the overlay row sum values becomes the dominant update cost. The worst-case update cost of the Basic Dynamic Data Cube becomes O(n) in the two-dimensional case.

We will improve upon this result in Section 4; first, however, we will present the general update cost formula of the basic tree for a d-dimensional data cube. As noted earlier, only one overlay box per tree level will be affected during an update. An overlay box at a given level contains exactly $(k^d - (k-1)^d)$ values that may be affected by an update. However, this formula may be approximated as (dk^{d-1}), which is strictly larger than $(k^d - (k-1)^d)$ for $d \geq 2$. This approximation formula is motivated by the observation that a given overlay box of dimensionality d has d sets of row sum values, and the size of each set is approximately k^{d-1}. At the root level, k=n/2, but the value of k decreases as we descend the tree; at each level of the tree, the value of k is divided in half. Thus, the cost of updating all the necessary overlay boxes during an update becomes the series

$$d(n/2)^{d-1} + d(n/4)^{d-1} + ... + d1^{d-1}$$

Rearranging terms, we have

$$= d[1^{d-1} + 2^{d-1} + 4^{d-1} + ... + (n/4)^{d-1} + (n/2)^{d-1}]$$

There are (log n) terms in this series. Substituting (n/2) as $((2^{\log n})/2)$ or $(2^{(\log n)-1})$,

$$= d[2^{0(d-1)} + 2^{1(d-1)} + 2^{2(d-1)} + 2^{3(d-1)} +... + 2^{((\log n)-2)(d-1)} + 2^{((\log n)-1)(d-1)}]$$

$$= d[(2^{d-1})^0 + (2^{d-1})^1 + (2^{d-1})^2 + (2^{d-1})^3 +... + (2^{d-1})^{(\log n)-2} + (2^{d-1})^{(\log n)-1}]$$

$$= d[(2^{d-1})^{\log n} - 1) / (2^{d-1} - 1)]$$

$$= d[(n^{d-1} - 1) / (2^{d-1} - 1)]$$

$$= O(n^{d-1})$$

In the next section, we present a modification to the basic tree that improves update performance; the resulting structure has sublinear complexity for both queries and updates.

4 Improving Updates

It is clear that storing overlay values directly in arrays results in costly update characteristics. As noted, the high update complexity of the overlay boxes is a consequence of dependencies between successive row sum values. Recall from Figure 7 that row sum values are cumulative sums of rows of cells covered by an overlay box. In Figure 13, an arrow illustrates the dependencies between cells in one set of row sum values. The value in row sum cell X_1 is a component of the value of cells $X_2..X_6$; therefore, when the value in cell X_1 changes, the values in cells $X_2..X_6$ are affected. Thus, an update to a single cell may cause a cascading update throughout the array. The series of dependencies between row sum values is at the heart of this update problem, and leads to the cascading updates that we have described.

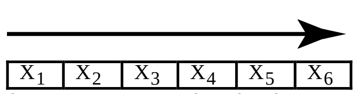

Figure 13. Dependencies between row sum values.

If we could remove or reduce the dependencies between row sums, perhaps the update cost for the tree as a whole can be significantly improved. The dependencies cannot be completely removed, however; the essence of the approach depends upon the existence of these dependencies, as illustrated in Figure 4. Instead, we propose a method of storing row sum values that ameliorates the series of dependencies between row sum values, and as a consequence attains efficient, balanced update and query characteristics for the tree as a whole. Our method takes a recursive approach, the recursion being with respect to the number of dimensions in the data cube. We first present an efficient means of handling the two-dimensional base case. We then present the method by which higher dimensional data cubes can be recursively reduced to two dimensions. We provide an inductive proof of the complexity of our approach which demonstrates that the tree provides sublinear complexity for both queries and updates.

4.1 The Two-Dimensional Case: The B^c Tree

We will analyze the two dimensional data cube as a special case of the d-dimensional data cube. We begin by examining the row sum values in the two dimensional data cube. An overlay for a two dimensional data cube has two sets of row sum values, each of which is one dimensional (Figure 6). Our goal is to reduce the cascading update that occurs when an individual row sum is updated. To this end, rather than store row sum values directly in an array, we will store them separately in an extension to the b-tree we call the *Cumulative B Tree* (B^c tree). There will be a separate B^c tree for each set of row sum values. The B^c tree is similar to a standard b-tree, with a few alterations. As in a standard b-tree, each node has a fixed maximum number of children; the maximum number of children per node is called the *fanout*. Each node stores keys associated with the children, and data is stored in the leaves of the tree.

Figure 14 shows a B^c tree for one set of row sum values in an overlay box. The B^c tree modifies the standard b-tree in two ways. The first modification is with regard to keys. Each leaf of the B^c tree corresponds to one row sum cell. For the purposes of insertion and lookup, 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 row sum values. Thus, the leaves of the B^c tree are in the same order as the row sum cells in the overlay box. Recall that row sum values are cumulative sums of rows; in the B^c tree, we store the sum of each individual row separately, and generate cumulative row sums as needed. The first leaf in the figure corresponds to the first row sum cell. Its key is thus 1, and it stores the value 14, which is the sum of the cells in the first row of the overlay box. The second leaf corresponds to the second row sum cell; its key is thus 2, and its value is (23-14=9), which is the sum of the cells in the second row of the overlay box. B^c trees also augment the standard b-tree by storing additional values in interior nodes. Along with the traditional pointer to each child, interior nodes of the B^c tree maintain subtree sums (STS). For each node entry, the 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 33, which represents the sum of the leaf values in the left subtree below the root (14+9+10). The interior node with key 3 has an STS of 9, which represents the sum of the leaf values in its left subtree (9).

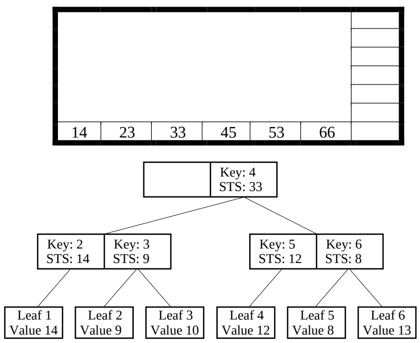

Figure 14. One set of row sum values stored in a B^c tree.

A row sum value is obtained from the B^c tree in O(log k) steps, where k is the number of row sum values in the overlay box. To calculate the row sum value for a given cell, traverse the tree using the cell's index as the key. Before descending to a node's child, the algorithm sums each preceding STS in the node. The following example makes use of the B^c tree shown in Figure 14. Suppose we wish to find the value of row sum cell 5 in the overlay box. We start at the root, using 5 as the key. 5 is in the right subtree of the root. The STS of 33 precedes it, so we add 33 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 (12 and 8). The STS 12 precedes the subtree we will descend, so we add it to our total. The STS 8 is after the subtree we will descend, so we ignore it. We descend to the leaf, which contains the value 8, and add it to our total, yielding 33+12+8=53. We are storing the sums of individual rows in the leaves of the tree, and thus the value we have calculated is the row sum value that is required for the overlay box. Assuming the tree fanout is f, a constant value, the worst-case query time of the B^c tree requires ($f*\log_f k$), or O(log k).

We next describe the algorithm for updating row sum values in a B^c tree. The update complexity is O(log k). For illustration, reconsider the B^c tree of Figure 14, and suppose an update to the data cube causes the row sum cell 3 to change from 10 to 15. We will update the B^c tree, and hence the row sum value, to reflect this change using 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 values is +5. We update the value of cell 3 with the new value (15). As we return up the tree, we will 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 did not fall in its left subtree. We next ascend to the root. As the changed cell falls within the left subtree of the root, we update the STS value in the root with the difference, yielding (33+5=38). At most one STS value will be modified per visited node during the update process, since we only update STS values corresponding to subtrees which contain the changed cell. Thus, updating the B^c tree requires O(log k). Using the B^c tree to store overlay box values in the two-dimensional case thus provides both query and update complexity of O(log k).

4.2 Storing Overlay Box Values Recursively

The B^c tree breaks the barrier to efficient updates of row sum values in one dimension. We now consider the general case, where the dimensionality of the data cube is greater than two. We have already noted that a two dimensional overlay box has two groups of row sum values, each of which is one dimensional. In general, an overlay box of d dimensions has d groups of row sum values, and each group is (d-1) dimensional (Figure 15). The row sum values of a three dimensional overlay consist of three planes, each of dimensionality two. We observe the fact that each group of row sum values has the same internal structure as array P. Recall that array P stores cumulative sums of cells in array A (Figure 3); row sum values store cumulative sums of rows within the overlay box. This concordance suggests that the two-dimensional row sum value planes be stored as two-dimensional data cubes using the techniques already described. 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; when d=2, we use the B^c tree to store the row sum values. Algorithms for query and update are as before, except that overlay box values are not accessed directly from arrays; rather, they are obtained from secondary trees.

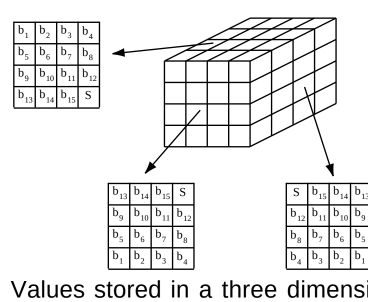

Figure 15. Values stored in a three dimensional overlay box (view is from lower rear).

4.3 Complexity of the Dynamic Data Cube

We now present an inductive proof of the performance complexity of the Dynamic Data Cube. We establish that the tree of trees has sublinear complexity for both queries and updates.

Theorem 1. Navigating a Dynamic Data Cube, not including the cost of accessing overlay box values in subtrees, requires $O(\log n)$, regardless of the dimensionality of the tree.

Proof of Theorem 1

A Dynamic Data Cube has $\log(n)$ levels; this follows from the manner in which overlay boxes recursively partition the data space. We descend exactly one child per node, which results in $\log(n)$ nodes being visited. Each node stores 2^d overlay boxes. One overlay box corresponds to the child that will be descended; no row sum values will be needed from this overlay box. We will require one row sum or subtotal value from each of the remaining $(2^d - 1)$ overlay boxes in the worst case, resulting in a maximum of $(2^d - 1)$ values accessed at any given level of the tree. As we have assumed that the dimensionality of a given data cube is fixed, this is a constant factor. Thus, ignoring the cost of accessing values in overlay boxes, we incur $O(\log n)$ to navigate a Dynamic Data Cube, irrespective of the dimensionality of the tree. ❑

Theorem 2. The complete Dynamic Data Cube, including subtrees, has query complexity of $O(\log^d n)$ and update complexity of $O(\log^d n)$.

Inductive Proof of Theorem 2

Base Case: Two dimensional tree

As noted earlier, overlay boxes are stored in B^c trees in the two-dimensional case. Thus, we must traverse individual B^c trees to obtain the necessary row sum values from overlay boxes in the primary tree. As shown earlier, the cost of traversing the B^c tree is $O(\log k)$, where k is the size of the overlay box. The size of the overlay box at the root of the primary tree is $k=(n/2)$, and the overlay box size grows geometrically smaller as we proceed down the levels of the primary tree towards the leaves. Thus, as we descend the primary tree, B^c trees are constructed for geometrically decreasing values of k. The cost of accessing each B^c tree therefore grows smaller as we approach the leaves of the primary tree. As noted earlier, in the worst case $(2^d - 1)$ row sum values will be required for each level of the primary tree. The cost of accessing all required B^c trees during a query is thus a series:

$$(2^2 - 1)\left[\log(n/2) + \log(n/2^2) + \log(n/2^3) + \dots + \log(2^4) + \log(2^3) + \log(2^2) + \log(2^1)\right]$$

This expression evaluates to

$$3\left[\log(n/2) + \dots + 4 + 3 + 2 + 1\right]$$
$$= (3)(^1/_2)(\log(n/2))(\log(n/2)+1)$$
$$= (3)((^1/_2)(\log^2(n/2)) + (^1/_2)(\log(n/2)))$$
$$= O(\log^2(n/2))$$

Thus, the total cost of accessing overlay box values in the two dimensional case is $O(\log n)$ for navigating the primary tree plus $O(\log^2(n/2))$ for the B^c trees which store the overlay box values, thus yielding $O(\log^2(n/2))$ for the complete structure. Note that the B^c tree has balanced query and update complexities; accordingly, the complexity of $O(\log^2(n/2))$ is also the update cost of the complete Dynamic Data Cube in the two-dimensional case. Therefore, the complexity for both queries and updates in the two-dimensional case is $O(\log^2 n)$, which is $O(\log^d n)$.

Inductive Case: $d > 2$

We make the inductive hypothesis that the complexity of a d-dimensional tree is $O(\log^d n)$, and show this implies that the complexity of a (d+1)-dimensional tree is $O(\log^{d+1} n)$. Using Theorem 1, the cost of queries and updates for a (d+1) dimensional tree is $O(\log n)$, plus the cost of accessing the overlay box values. Recall that, when $d>2$, overlay boxes are stored in their own separate Dynamic Data Cubes, each of dimensionality (d-1), which we refer to as *secondary* trees. In this case, since the primary tree has dimensionality of (d+1), the secondary trees have dimensionality d; therefore, by the inductive hypothesis, each secondary tree has complexity $O(\log^d n)$. There will be $(2^{d+1} - 1)$ secondary trees accessed at each level of the primary tree, a constant factor. The cost of accessing the complete (d+1) dimensional tree is thus $(2^{d+1} - 1)(\log^d n)(\log n)$, or $O(\log^{d+1} n)$. Therefore, a (d+1) dimensional tree has query and update complexity of $O(\log^{d+1} n)$. ❑

4.4 Discussion

Storage requirements are often an important factor when evaluating new methods. The space required to store array A is n^d cells. The leaf level of the tree stores array A, and therefore also requires n^d cells of storage. Higher levels of the tree, however, require increasingly less space to store. Recall that each overlay box requires exactly $(k^d - (k-1)^d)$ cells of storage, and that k doubles for each successively higher level of the tree (i.e., each tree level has its own value of k). Table 2 presents a comparison between the storage requirements of overlay boxes versus the storage required by the corresponding covered region in array A. As k increases, the overlay box storage, as a percentage of the region it covers, decreases dramatically. From Table 2, and also from Figure 9, it is apparent that most of the additional storage required by the Dynamic Data Cube is found in the lowest levels of the tree. In contrast, levels of the tree closer to the root occupy considerably less space. This trend holds regardless of the dimensionality of the tree.

Table 2. Required storage, overlay boxes versus array A.

k	d	Overlay Box $= k^d - (k-1)^d$	Region in A $= k^d$	Percentage O.B. / A
2	2	3	4	75.00%
4	2	7	16	43.75%
8	2	15	64	23.44%
64	2	127	4096	3.10%
256	2	511	65536	0.78%

We therefore propose the following optimization to the tree to conserve space and improve overall performance. We will delete a given number of the lowest, most space-consuming levels of the tree immediately above the leaves. Let the leaves store array A directly as before. We define the level of the tree immediately above the leaves as *tree level 1*. Starting at tree level 1 and working upwards towards the root, we will delete h tree levels. After these tree levels have been removed, the new tree level 1 will not have an overlay box size of k=2 as before, but will contain overlay boxes of size $k=2^{h+1}$. Higher levels of the tree would remain as before. Note that we are not changing the fanout or the essential tree structure; we are merely eliminating h levels of the tree immediately above the leaf level, and consequently conserving the storage space those levels would have consumed. Since the lowest tree levels are dense, their elimination results in considerable space savings. By setting the appropriate value of h, one can reduce the storage required by the Dynamic Data Cube to within ε of the size of array A.

This optimization induces an associated performance cost. We have eliminated h levels of overlay boxes which provided partial region sums of the target region. When we reach the leaf level, we will have to sum leaf cells to calculate the sum of the missing region. For example, refer to Figure 11. Suppose we eliminate one tree level in the figure by setting h=1; thus, tree level 1 would be deleted, to be replaced by tree level 2. Deleting the level saves 48 cells of storage, or 34%. As a result, however, the region sums provided by boxes **U**, **V**, **W**, and **Z** would no longer be available. To compensate for these missing region sums, during queries we will have to sum the appropriate leaf cells to calculate the sum of any missing region. We note that, given any individual query, all deleted regions will be adjacent at the leaf level; furthermore, the maximum size of the union of these deleted regions is $2^{(h+1)d}$ leaf cells. Therefore, in the worst case, this optimization would require the addition of $2^{(h+1)d}$ adjacent leaf cells when the query reaches the leaf level. This cost is offset by the fact that the deletion of tree levels will have a positive impact on tree traversal times, since the number of levels in the tree affects the number of accesses to secondary storage during traversal. The appropriate value of h for a given application would be determined by balancing the desired space savings and the tree traversal time savings against the cost of the additional computation required by the optimization.

Very high dimensionality of the cube (e.g., d>20) will present performance hurdles. This "dimensionality curse" is a well-known problem in the field of multidimensional database indexes, and affects all known methods. Limiting the number of dimensions in the cube is currently the best option for system designers; see the work by Harinarayan et. al. for a discussion of this topic [HRU96]. The performance characteristics of the Dynamic Data Cube nevertheless allow the incremental construction and maintenance of dramatically larger data cubes, at higher dimensionality, than other methods.

5 Dynamic Growth of the Data Cube

The prefix sum and relative prefix sum methods do not address the growth of the data cube; instead, they assume that the size of each dimension is known a priori. For many potential applications, however, it is more convenient to grow the size of the data cube dynamically to suit the data. For example, astronomers who are analyzing stars might form a data cube for their star database. They expect to discover more stars in the future. Clearly it would not be efficient to create a data cube that initially contains cells for all possible locations of star systems in the Universe, particularly since the vast majority of the resulting cells would always be empty. Rather, it is more practical to create the data cube initially only for locations of existing star systems; as additional systems are discovered, new cells can be added to the data cube. New star systems, however, can be found in any direction relative to existing systems, therefore the data cube must be able to grow in any direction relative to its existing cells. The direction of data cube growth should be determined by the data, and not a priori. The capability to grow the data cube dynamically in any direction (i.e., not merely appending to a single edge of the data cube) is very important in many application environments.

This example also illustrates another issue. In many application domains data is essentially clustered, and there are large unpopulated regions in the data space. Consider the case of NASA's EOSDIS satellites, which generate an enormous volume of data every day. The data is principally in the form of measurements of numerous environmental variables on the Earth, such as rates of vegetation growth, rates of methane gas production, etc. Measurements are made for the entire surface of the planet, yet the data is essentially clustered; for example, methane gas production is largely concentrated around agricultural and industrial centers. There are vast, unpopulated regions of the data space; for example, methane gas production may be essentially zero over oceans. This information is not static, however, and new point sources of methane gas production may arise, such as when new cattle ranches or factories comes on-line in previously undeveloped areas. Range sum queries over a data cube formed from such data would be very useful, providing scientists with aggregate measurements for any arbitrary region of the globe.

Neither the prefix sum method nor the relative prefix sum method gracefully handle these situations. There are several difficulties. Neither approach makes any provision for empty or non-existent regions of cells within the data cube. Figure 16 shows an example of a cell, denoted *, being added to an existing data cube. Since empty regions are not allowed with these methods, the creation of cell * forces the further creation of all cells in the shaded region. This results in the first difficulty for these methods: since they must store all cells in the range of each dimension, a significant amount of storage space may be wasted for regions that are unpopulated. A more serious consequence directly follows. For correctness of later queries, these methods would require that the values of cells in the shaded region be computed and stored when cell * is added.

Furthermore, in a dynamic environment new cells may be added in any direction relative to existing cells. When the new cell precedes cells in the existing data cube, as in Figure 17, those cells must also be updated. This fact results from the cell dependencies inherent in these approaches, i.e., all cell values are dependent on the values of cells that precede them. In the worst case, when a new cell is created that precedes all existing cells, every cell in the array will require updating; thus, the creation of a new cell may incur worst-case update performance cost for these methods. In consideration of these properties, we suggest that the prefix sum approach can be extended to support efficient growth of the data cube, as long as growth occurs within certain constraints: growth should only occur at the rightmost edges of the data cube, e.g., append-only growth; and, all cells in newly-created rows should be added at the same time, to avoid the situation illustrated in Figure 16.

Index	0	1	2	3	4	5	6	7	8	9	10
0	3	8	9	11	13	17	23	26			
1	10	18	21	29	39	50	57	62			
2	12	24	29	40	53	67	78	88			
3	15	29	35	51	67	86	99	117			
4	19	35	42	61	80	103	123	142			
5	21	40	50	75	95	126	151	172			
6	25	49	61	93	114	154	182	206			
7	27	55	69	103	127	168	205	230			
8											
9											*

Figure 16. Creation of a new cell after existing cells.

			3	8	9	11	13	17	23	26
			10	18	21	29	39	50	57	62
*			12	24	29	40	53	67	78	88
			15	29	35	51	67	86	99	117
			19	35	42	61	80	103	123	142
			21	40	50	75	95	126	151	172
			25	49	61	93	114	154	182	206
			27	55	69	103	127	168	205	230

Figure 17. Creation of a new cell before existing cells.

The Dynamic Data Cube is well suited to dynamic growth of the data cube due to the properties of the overlay box. Each overlay box covers some region in array A; when all the cells in that region of A are equal to zero, the overlay box values will also be equal to zero. We therefore need not store overlay boxes or subtrees associated with regions that are completely empty, since all row sums from such regions are zero. Accordingly, we can build the data cube incrementally over time using a Dynamic Data Cube. We begin with one node, the root, of size 2^d cells. Suppose information is to be added that falls in a cell that is outside the boundaries of the root; we call this *adding a new cell* (Figure 18a). When a new cell is added, we create a new root above the current root, thereby creating a new tree level. The new root has a size that is twice the size of the previous root in each dimension. The old root is placed as a child of the new root, and overlay box information is generated for it (Figure 18b). Data cube growth can occur in any direction. The parent grows the data space towards the new cell. Thus, the placement of the previous root in its new parent depends on the location of the new cell; to avoid certain forms of pathological growth, a round-robin approach may be used when resolving ambiguity. We do not allocate storage for the new sibling nodes of the previous root, however, since overlay boxes for these empty regions will have row sums that evaluate to zero. In the figures, the shaded areas represent regions for which no storage is allocated.

Level 0 (Old root)

3	5	* 4
7	3	

Figure 18a. Adding a new cell.

Level 1: New root

	8	*	
10	18		

Level 0

3	5	* 4	
7	3		

Figure 18b. New root created.

Level 1

	8	*	4
10	18	4	4

3	5	* 4	0
7	3	0	0

Figure 18c. New sibling.

We continue to create new roots in this manner, doubling the size in each dimension for each successive root, until a root is created that encompasses the new cell. At this point, we traverse down the tree to the new cell, creating child nodes and associated overlay boxes as we descend (Figure 18c). Recall that 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. No overlay box or subtree is instantiated for regions that are entirely empty. Should we encounter a non-existent overlay box during a query, we may conclude that its row sum values are all zero. We note that this method of tree growth may result in the construction of trees of non-optimal height when data is inserted in certain pathological distributions and sequences. While a complete discussion of this topic is beyond the scope of this paper, we will observe that, since each newly-created root is double the size of the previous root, this sub-optimal effect would be naturally limited when trees are built using real data.

The B^c tree also supports incremental construction. Being derived from the standard b-tree, it gracefully handles growth of its data. As with the Dynamic Data Cube, we need not create all nodes in the B^c tree at instantiation. Even when a query seeks a cell that has not yet been inserted in the B^c tree, the B^c tree will still return the correct row sum value associated with that cell. Therefore, the B^c tree also supports incremental construction of the Dynamic Data Cube.

This incremental construction of the Dynamic Data Cube is naturally suited to clustered data and data that contains large, unpopulated 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.

6 Conclusion

We present the Dynamic Data Cube, a new method for handling range sum queries in data cubes. We describe the construction of the Basic Dynamic Data Cube. We develop the B^c tree, which breaks the update complexity barrier, and demonstrate its use in the construction of the recursive Dynamic Data Cube. We further present a method of constraining the space requirements of the Dynamic Data Cube to within ε of the full data cube size by deleting unnecessary tree levels. We discuss the properties of the Dynamic Data Cube which enable it to handle sparse and clustered data, as well as empty regions of the cube, efficiently.

The Dynamic Data Cube provides an efficient means of storing and maintaining data cubes for a wide variety of emerging applications. Table 3 presents the performance complexities of various methods of computing range sum queries. Our analysis reveals the impact of the dimensionality of the data cube on performance by referring to n, which is the size of the data cube in each dimension, rather than N, the total size of the data cube; for reference, $N=n^d$. The Dynamic Data Cube has performance complexity of $O(\log^d n)$ for both queries and updates. It provides the capability to grow the data cube dynamically in any direction, and the ability to handle sparse and clustered data gracefully. As noted in the discussion of Table 1, these characteristics significantly lower the barriers to the adoption of data cube methods in novel application domains.

Table 3. Performance complexities of various methods.

Method	Performance for input size N ($N=n^d$)	
	Query	Update
Naive approach	$O(n^d)$	$O(1)$
Prefix Sum [HAMS97]	$O(1)$	$O(n^d)$
Relative Prefix Sum [GAES99]	$O(1)$	$O(n^{d/2})$
Dynamic Data Cube	$O(\log^d n)$	$O(\log^d n)$

References

[AAD+96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, pages 506-521, Mumbai (Bombay), India, September 1996.

[AGS97] 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.

[Cod93] E. F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: an IT mandate. Technical report, E.F. Codd and Associates, 1993.

[GAES99] S. Geffner, D. Agrawal, A. El Abbadi, T. Smith. Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes. To appear in *Proc. of the 15th International Conference on Data Engineering*, Sydney, Australia, March 1999.

[GBLP96] 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.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, J. Ullman. Index selection for OLAP. In *Proc. of the 13th Int'l Conference on Data Engineering*, Birmingham, U. K. April 1997.

[HAMS97] 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.

[HRU96] 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.

[JS96] T. Johnson, D. Shasha. Hierarchically split cube forests for decision support: description and tuned design, 1996. Working Paper.

[OLA96] The OLAP Council. *MD-API the OLAP Application Program Interface Version 5.0 Specification*, September 1996.

[SDNR96] A. Shukla, P. M. Deshpande, J. F. Naughton, K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, pages 522-531, Mumbai (Bombay), India, September 1996.

[SR96] B. Salzberg, A. Reuter. Indexing for aggregation, 1996. Working Paper.

[VWI98] J. S. Vitter, M. Wang, B. Iyer. Data Cube Approximation and Histograms via Wavelets. In *Proceedings of the 1998 ACM Seventh International Conference on Information and Knowledge Management (CIKM'98)*, pages 96--104.

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

This research is patent pending. For licensing information, contact Mathew Grell at the University of California Office of Research.