Space-Efficient Data Cubes for Dynamic Environments

Mirek Riedewald[1], Divyakant Agrawal[1], Amr El Abbadi[1], and Renato Pajarola[2]

[1] Department of Computer Science, University of California, Santa Barbara CA 93106, USA
{mirek, agrawal, amr}@cs.ucsb.edu
[2] Department of Computer Science, University of California, Irvine CA 92697, USA
pajarola@acm.org

Abstract. Data cubes provide aggregate information to support the analysis of the contents of data warehouses and databases. An important tool to analyze data in data cubes is the range query. For range queries that summarize large regions of massive data cubes, computing the query result on the fly can result in non-interactive response times (e.g. in the order of minutes). To speed up range queries, values that summarize regions of the data cube are pre-computed and stored. This faster response time results in more expensive updates and/or space overhead. While the emphasis is typically on low query and update costs, growing data collections increase the demand for space-efficient approaches. In this paper two techniques are presented that have the same update and query costs as earlier approaches, without introducing any space overhead.

Key Words: MOLAP, Data Cube, Data Warehousing, Online Aggregation, Summarization

1 Introduction

Data cubes, also known as multidimensional databases, are powerful tools to support the analysis of the contents of data warehouses and databases. A data cube is similar to a multidimensional array. Certain attributes of the database are chosen to be *measure attributes*. These are the attributes whose values are of interest to an analyst. Other attributes are selected as *dimensions* (also called *functional attributes*). The measure attributes are aggregated according to the dimensions. A *cell* of the data cube is described by a unique combination of dimension values. An example of a data cube based on the TPC-H benchmark database [2] would have the total price of an order as the measure attribute and the region of a customer and the order date as the dimensions. Such a data cube provides the aggregated total orders for all combinations of regions and dates. Queries issued by an analyst who wants to examine how the customer behavior in different regions changes over time (e.g., in order to evaluate the success of local advertising campaigns) do not need to access and join the "raw" data in the different tables. Instead the information is available and summarized from the data cube. Note that our data cube notion differs from the terminology used in [6]. We do not augment the data cube with pre-computed results of GROUP-BYs of subsets of the set of all dimension attributes. Thus our data cube notion corresponds to the data cube *core* in [6].

Aggregate range queries are useful analysis tools on data cubes. Such a range query aggregates the values of those cells that satisfy the range selection condition for all dimensions. For instance, a range query on our example data cube could "Find the total amount of orders in California over the last four months". Queries of this form are useful in discovering relationships between attributes in the database. In this paper we concentrate on the aggregation operator SUM that computes the sum of the values of the selected cells. The approach can be generalized to other operators like COUNT, Average, Rolling sum, etc.

Analyzing data online is a highly interactive process. Analysts expect fast responses to their queries, ideally in the order of seconds at most. For massive data sets, however, range queries that access and aggregate on the fly the contents of a large number of cells, will show slow response times. To speed up those queries, the aggregates for sets of cells are pre-computed and stored in the data cube. This leads to well-known tradeoffs. Storing additional pre-computed values results in space overhead. Also, updates become more expensive when an update to a single cell triggers updates to all pre-computed values that include this cell in their aggregation. Different applications tolerate different update costs. While *what-if* scenarios and stock trading applications require fast updates, for other applications overnight batch processing of updates suffices. But even batch processing benefits from faster updates, since they reduce the size of the update window and allow for more frequent updates and shorter inaccessibility of the data. Ideally a data cube should support fast queries and fast updates at no extra storage cost.

An elegant algorithm for computing range queries that return the sum of the selected cells in data cubes is presented in [7]. We refer to it as the *Prefix Sum* technique (PS). The essential idea is to pre-compute the prefix sums of the data cube (see Figure 1), which are used to answer ad hoc queries in constant time. Since the prefix sums replace the original values in the cells, the PS technique does not require additional space. The approach is mainly hampered by its update costs. In the worst case an update to a single cell requires recomputing the whole array, which is of the same size as the original data cube.

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

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

To reduce the high update costs, while still guaranteeing a constant query cost, the *Relative Prefix Sum* technique (RPS) [4] controls the cascading updates. This comes at the cost of a space overhead. In contrast, the *Hierarchical Cubes* techniques (HC) [1] do not require additional space. They generalize the idea of RPS by allowing different tradeoffs between update and query cost. The tradeoff is selected by setting parameters that control the generation of the pre-computed values. Consequently the query and update costs depend on those parameters as well as the dimensionality and the size of the data cube. This makes a general comparison of HC to the other techniques difficult. For instance, while for some data cubes one of the HC techniques might provide a parameter setting that leads to a better query and update behavior than RPS, for other data cubes this will not be the case.

The only technique that guarantees that query and update cost are both sublinear in the domain size of the dimensions for any data cube is the *Dynamic Data Cube* (DDC) [3]. The space overhead of this technique, however, is significant.

For massive data sets the space requirements of a technique become a decisive factor. Space overhead not only leads to extra costs for storage devices. The additional values also cause additional propagations of updates and longer access times on the physical devices.

In this paper we present two new space-efficient data cube techniques – SRPS and SDDC – based on RPS and DDC, respectively. Both techniques inherit the update and query costs of their predecessors, but considerably reduce the space requirements. More precisely, they have the same storage consumption as the original data cube, thus do not introduce any space overhead. They are suitable for data warehousing environments, especially decision support and OLAP applications. SRPS efficiently supports applications where queries dominate. SDDC balances the costs of queries and updates. Thus it is especially appropriate in settings with frequent updates and enables users to analyze what-if scenarios.

In Section 2 we describe the SRPS and SDDC techniques. Both techniques are compared to their predecessors RPS and DDC, respectively. Section 3 concludes this article.

2 SRPS and SDDC: Two Space-Efficient Data Cube Techniques

In this section SRPS and SDDC are presented and compared to their "predecessors" RPS and DDC, respectively. The following notation will be used. Let A be a data cube of dimensionality d, and let $c = (c_1, \ldots, c_d)$ be a cell that contains the value $A[c]$.[1] Without loss of generality let the domain of each dimension attribute i be $\{0, 1, \ldots, n_i\}$. $e : f$ is a *region* of the data cube, more precisely the set of all cells c that satisfy $e_i \leq c_i \leq f_i$ for all $1 \leq i \leq d$ (i.e., $e : f$ is a hyper-rectangular region of the data cube). Cell e is the *anchor* and cell f the *endpoint* of the region. Consequently the entire data cube is anchored at $(0, \ldots, 0)$ and ends at $(n_1 - 1, \ldots, n_d - 1)$. The set of the values in region $e : f$ is denoted $A[e] : A[f]$, and $\mathrm{op}(A[e] : A[f])$ is the result of applying the aggregate operator op to those values.

SRPS and SDDC make use of the *inverse property* of some aggregation operators. They 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. SUM, COUNT). For the SQL operator SUM (sum of the values of the selected cells) each region's sum can be obtained by adding and subtracting sums for appropriate regions that are anchored at $(0, \ldots, 0)$ (see Figure 2 for an example). Note that according to [7] any range sum can be computed by combining up to 2^d (which is a constant) sums for ranges anchored at $(0, \ldots, 0)$. Thus the problem of computing the sum for an arbitrary range is reduced to the problem of efficiently computing the sum for any region that is anchored at $(0, \ldots, 0)$. We will therefore only describe how SRPS and SDDC solve this problem.

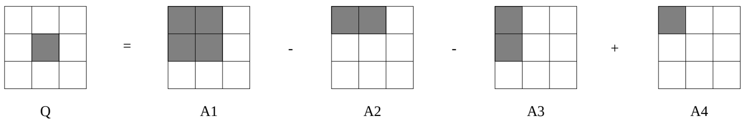


Fig. 2. A geometric illustration of the two-dimensional case: SUM(Q) = SUM(A1) - SUM(A2) - SUM(A3) + SUM(A4)

For the sake of clarity let each of the d dimensions have a domain of size n. The SRPS and SDDC techniques are described for the aggregate operator SUM. Other operators for which exists an inverse operator can be handled in a similar way. In our analysis query and update costs are expressed in terms of the number of accessed cells of the data cube. The storage cost is measured in terms of cells as well.

2.1 RPS: The Relative Prefix Sum Technique

In this section we give an overview of the RPS technique [4]. Note that the analysis of the update and query costs in [4] is not correct for data cubes with more than two dimensions.

[1] Note that each c_i is an element of the domain of the corresponding dimension, and that $A[c]$ is the corresponding value of the measure attribute(s).

This, however, does not affect the asymptotic costs, but rather the constants in the formulas. A correct analysis for RPS can be found in [5].

Like the Prefix Sum technique, RPS makes use of the inverse property of some aggregate operators. Thus the problem of summarizing any possible range is reduced to the problem of summarizing and combining ranges that are anchored at $(0, \ldots, 0)$. The main idea of RPS is to avoid the cascading updates of the Prefix Sum technique by dividing the data cube into smaller chunks of equal size, called *overlay boxes*. The prefix sums are computed and stored relative to the anchor cell of an overlay box. The array with those relative prefix sums has the same size as the original data cube. Since the relative prefix sums only provide aggregate information about the cells *inside* the overlay box, an additional data structure – the *overlay array* – is used. The overlay array provides sums for regions of cells *outside* the overlay boxes. Together the overlay and the relative prefix sum array guarantee a worst case query cost of 2^d for ranges anchored at $(0, \ldots, 0)$, a worst case query cost of 2^{2d} for general ranges, and a worst case update cost of $(2\sqrt{n} - 2)^d$. Compared to directly storing the original data cube, the RPS technique incurs a space overhead of the size of the overlay array. Depending on the parameters (dimensionality, size of the data cube and the overlay boxes) this overhead ranges from a few percent up to almost 100% of the data cube size in the worst settings. In Figure 3 the overlay array and the array with the relative prefix sums are displayed for the data cube A from Figure 1.

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

Fig. 3. Overlay (OL) and relative prefix sum (RP) arrays with overlay boxes drawn for reference

2.2 SRPS: The Space-Efficient Relative Prefix Sum Technique

Like the Relative Prefix Sum method, SRPS provides constant-time queries with an update complexity of $O(n^{d/2})$, compared to an update cost of $O(n^d)$ for the Prefix Sum technique. SRPS improves on RPS by not needing the additional overlay array and thus removing the space overhead. Due to this improvement the Prefix Sum technique becomes a special case of SRPS (reasons are given in the next section).

Description of the Technique The data cube is completely partitioned into a set of disjoint hyper-rectangles of equal size. We will refer to those hyper-rectangles as *boxes*. For clarity and without loss of generality let the length of a box in each dimension be k. Thus the data cube is partitioned into $\lceil n/k \rceil^d$ boxes of size k^d each. The boxes are anchored at cells whose coordinates each are multiples of k.

Let B be a box that is anchored at cell $a = (a_1, \ldots, a_d)$. Then box B contains all cells c that satisfy $a_i \leq c_i < a_i + k$ for all $1 \leq i \leq d$. The box cells on the "upper left" surfaces, i.e., all those cells that agree with the anchor cell a in at least one coordinate, are referred to as *border cells*. The other cells, i.e., those cells c with $a_i + 1 \leq c_i < a_i + k$ for all $1 \leq i \leq d$, are *inner cells*. Essentially inner cells only store sums local to the box, while border cells include cells from outside the box into their aggregation.

Any box cell c stores the value $\text{SUM}(A[l_1, l_2, \ldots, l_d] : A[c])$ where

$$\forall i : 1 \leq i \leq d \quad \begin{cases} l_i = 0 & , \text{if} \quad c_i = a_i \\ l_i = a_i + 1 & , \text{if} \quad a_i + 1 \leq c_i < a_i + k \end{cases}$$

Note that the above definition covers all possible cases for c_i since c is contained in the box anchored at a. Further note, that border cells for at least one dimension i satisfy $c_i = a_i$, while for inner cells the second inequality ($a_i + 1 \leq c_i < a_i + k$) holds for all dimensions. We will use the term *aggregation region* for the described regions of cells. Thus for example the anchor cell a stores $\text{SUM}(A[0, \ldots, 0] : A[a])$, i.e., has the aggregation region $A[0, \ldots, 0] : A[a]$. The other border cells aggregate hyper-rectangular regions of cells that stretch from the surface of the box to the corresponding surface of the data cube. Inner cells c store $\text{SUM}(A[a_1+1, a_2+1, \ldots, a_d+1] : A[c])$, which is the prefix sum relative to cell $(a_1 + 1, a_2 + 1, \ldots, a_d + 1)$. Figure 4 shows aggregation regions (shaded) for a two-dimensional data cube. Note that border values in the same surface are cumulative, e.g., border cell b2's aggregation region contains the aggregation region of b1. In the example in Figure 5 the original data cube[2] and the corresponding SRPS cube are shown.

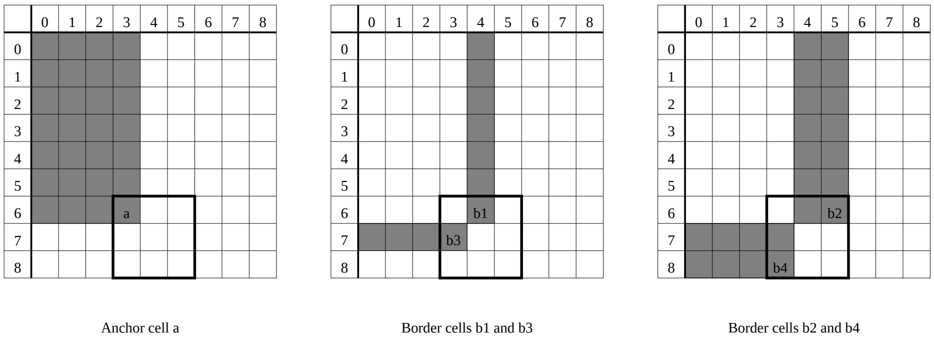


Fig. 4. Computation of border values as the sum of the values of the cells in the shaded area on array A

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

	0	1	2	3	4	5	6	7	8
0	3	5	6	11	2	6	23	3	6
1	7	3	5	18	8	15	34	2	6
2	9	7	11	29	11	21	55	7	18
3	15	14	20	51	16	35	99	18	34
4	4	2	3	10	3	7	24	1	4
5	6	5	9	24	4	16	52	2	6
6	25	24	36	93	21	61	182	23	47
7	2	4	6	10	3	4	23	1	4
8	7	8	13	23	6	9	42	10	19

Fig. 5. Original data cube (left) and corresponding SRPS cube with boxes of size 3×3 (right)

[2] It is the same cube as the one in Figure 1.

SRPS by definition does not cause any space overhead. All pre-computed values "fit" into an array of the size of the data cube. Once the SRPS cube is constructed, the original data cube can be discarded. All queries and updates are directed to the SRPS cube. The space savings compared to the RPS technique can be considerable. For RPS a prefix array of size n^d (i.e., size of the original data cube) and an overlay array of size $(n/k)^d(k^d - (k-1)^d) = n^d(1 - (\frac{k-1}{k})^d)$ were stored. Thus SRPS saves storage of the size $n^d(1 - (\frac{k-1}{k})^d)$.

Due to the improvements, the Prefix Sum [7] technique now constitutes a special case of SRPS. Namely, when setting $k = 1$ the boxes collapse to single cells that store the corresponding prefix sum.

Querying SRPS As mentioned earlier, we describe the query algorithm only for range queries that are anchored at $(0, \ldots, 0)$. The range sum for such a query can be obtained by adding values from a single box.

Assume that a query computes the sum for the range $(0, \ldots, 0) : q$ where $q = (q_1, \ldots, q_d)$ is the endpoint of the query region. This query has to compute the value of $\mathrm{SUM}(A[0, \ldots, 0] : A[q])$. Let q be inside box B which is anchored at cell a, i.e. $a_i \leq q_i < a_i + k$ for all $1 \leq i \leq d$. The region $A[0, \ldots, 0] : A[q]$ can be partitioned into the following 2^d non-intersecting regions.

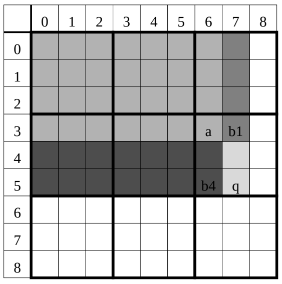
0. $A[0, \ldots, 0] : A[a]$
1. $A[a_1 + 1, 0, \ldots, 0] : A[q_1, a_2, \ldots, a_d]$, $A[0, a_2 + 1, 0, \ldots, 0] : A[a_1, q_2, a_3, \ldots, a_d]$,..., $A[0, \ldots, 0, a_d + 1] : A[a_1, \ldots, a_{d-1}, q_d]$
2. $A[a_1 + 1, a_2 + 1, 0, \ldots, 0] : A[q_1, q_2, a_3, \ldots, a_d]$,
   $A[a_1 + 1, 0, a_3 + 1, 0, \ldots, 0] : A[q_1, a_2, q_3, a_4, \ldots, a_d]$,...,
   $A[a_1 + 1, 0, \ldots, 0, a_d + 1] : A[q_1, a_2, \ldots, a_{d-1}, q_d]$,
   $A[0, a_2 + 1, a_3 + 1, 0, \ldots, 0] : A[a_1, q_2, q_3, a_4, \ldots, a_d]$,...,
   $A[0, a_2 + 1, 0, \ldots, 0, a_d + 1] : A[a_1, q_2, a_3, \ldots, a_{d-1}, q_d]$,...,
   ..., $A[0, \ldots, 0, a_{d-1}, a_d + 1] : A[a_1, \ldots, a_{d-2}, q_{d-1}, q_d]$

   $\vdots$

d. $A[a_1 + 1, a_2 + 1, \ldots, a_d + 1] : A[q_1, q_2, \ldots, q_d]$

If $q_i = a_i$ for any i $(1 \leq i \leq d)$, then those of the above regions that contain the index value $a_i + 1$ are empty. Especially, if $q = a$, i.e., the query ends at the anchor cell of the box, then all regions except $A[0, \ldots, 0] : A[a]$ will be empty.

Figure 6 shows an example for the partitioning of the query region for a two-dimensional data cube. SRPS is constructed such that the above partitions of the query region match the aggregation regions of the corresponding cells that have the same coordinates as the endpoint of the partition. Thus the aggregate sum for the query region can be computed by adding the corresponding pre-computed values. The cells that store the values are simply obtained as the "projection" of cell q to the surfaces of box B that contain the border cells (including cell q itself). More precisely, the set of all cells c that have to be accessed in order to compute $\mathrm{SUM}(A[0, \ldots, 0] : A[q])$ is $\{c = (c_1, \ldots, c_d) | \forall i : 1 \leq i \leq d \wedge c_i \in \{a_i; q_i\}\}$ (in Figure 6 the cells a, $b1$, $b4$ and q are accessed). Thus in the worst case 2^d cells need to be accessed to compute the range sum for a query that is anchored at $(0, \ldots, 0)$. Since an arbitrary range query can be answered by respectively adding and subtracting the results for up to 2^d queries that are anchored at $(0, \ldots, 0)$, the *overall worst case range query cost* for SRPS becomes 2^{2d}, which is independent of n and hence the query cost is constant irrespective of the size of the dimension domains of the data cube.



Partitioning of the query region

	0	1	2	3	4	5	6	7	8
0	3	5	6	11	2	6	23	3	16
1	7	3	5	18	8	15	34	2	6
2	9	7	11	29	11	21	55	7	18
3	15	14	20	51	16	35	99	18	34
4	4	2	3	10	3	7	24	1	4
5	6	5	9	24	4	16	52	2	6
6	25	24	36	93	21	61	182	23	47
7	2	4	6	10	3	4	23	1	4
8	7	8	13	23	6	9	42	10	19

Query result: 99+18+52+2=171

Fig. 6. Partitioning of the query region; and cells that contain the pre-computed sum for the partitions

Updating SRPS In general an update to a single cell affects all those cells that store a pre-computed value that depends on that cell. Figure 7 shows two examples. An update to the cell marked with * has to be propagated to each of the shaded cells.

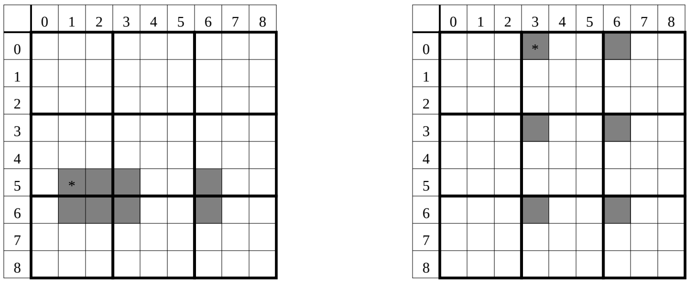


Fig. 7. Cells affected by an update to (5,1) (left) and cells affected by an update to (0,3) (right)

To keep the description of the analysis of the update cost simple, we assume that k, the side-length of each box, evenly divides n, the side-length of the data cube. Also, recall that a cell c which is contained in a box that is anchored at cell a stores the value $\mathrm{SUM}(A[l_1, l_2, \dots, l_d] : A[c])$ where

$$\forall i : 1 \leq i \leq d \quad \begin{cases} l_i = 0 & \text{, if} \quad c_i = a_i \\ l_i = a_i + 1 & \text{, if} \quad a_i + 1 \leq c_i < a_i + k \end{cases}$$

i.e., $(l_1, l_2, \dots, l_d) : c$ is c's aggregation region. In dimension i ($1 \leq i \leq d$) c's aggregation region ranges over the coordinates from l_i to c_i. According to the above definition there are two possible coordinate ranges in dimension i. If $c_i = a_i$, then the aggregation region ranges from 0 to a_i. Otherwise it ranges from $a_i + 1$ to c_i. For the sake of a more concise description we will use $\langle x, y \rangle$ to denote the set $\{x; x+1; x+2; \dots; y\}$.

Let $u = (u_1, \dots, u_d)$ be the updated cell. Clearly the number of cells c that are affected by an update to u is equal to the number of aggregation regions that contain u. Thus, in order to obtain the update cost, we determine how many cells in the SRPS cube contain the updated cell u in their aggregation region. Let cell c be contained in the box that is anchored at cell a. Then c's aggregation region in dimension i includes either the coordinates in the range $\langle 0, c_i \rangle$ (if $c_i = a_i$) or the coordinates in the range $\langle a_i + 1, c_i \rangle$ (if $a_i + 1 \leq c_i < a_i + k$). There are n/k coordinate ranges of the type $\langle 0, c_i \rangle$, because there are n/k possible values for a_i (anchor coordinates are multiples of k). Each of those ranges could contain u_i. Also, for any u_i there is exactly one a_i such that $a_i \leq u_i < a_i + k$. Therefore there are at most $k-1$ coordinate ranges of the type $\langle a_i+1, c_i\rangle$ that could contain u_i (namely the ranges $\langle a_i+1, a_i+1\rangle$, $\langle a_i+1, a_i+2\rangle$,...,$\langle a_i+1, a_i+k-1\rangle$). Consequently, in total there are at most $(n/k)+(k-1)$ coordinate ranges in dimension i that could contain u_i. Since this holds for all dimensions, we obtain an upper bound of $(n/k+k-1)^d$ on the number of aggregation regions that could contain the updated cell, and thus on the update cost.

To tighten this bound, we make the following observation. If $u_i = 0$, then u_i can not be contained in any coordinate range of the type $\langle a_i+1, c_i\rangle$. The number of coordinate ranges that could contain $u_i = 0$ is thus reduced to n/k. Otherwise, i.e. if $u_i > 0$, there are at most $n/k-1$ anchor coordinates that are greater or equal than u_i (since there are n/k possible anchor coordinates and $a_i = 0$ is one of them). This reduces the number of coordinate ranges that could contain $u_i > 0$ to $(n/k-1)+(k-1)$. Except for $k = 1$, which lets SRPS collapse to the Prefix Sum method, the second case ($u_i > 0$) represents the worst case, because $n/k+k-2 \geq n/k$. The upper bound on the update cost thus improves to $(n/k+k-2)^d$. This bound is tight; it is met when cell $(1, 1, \ldots, 1)$ is updated. Note, that the aggregation regions that contain the updated cell are well defined by the above formulas for the coordinate ranges. The set of cells that need to be updated is easily obtained as the set of endpoints of those regions.

The update costs are minimal for $k = \sqrt{n}$, resulting in a *worst case update cost* of $(2\sqrt{n}-2)^d = \mathrm{O}(n^{d/2})$. Changing k does not affect the worst case query costs.[3] Consequently, choosing $k = \sqrt{n}$ results in the optimal SRPS cube.

2.3 DDC: The Dynamic Data Cube Technique

In this section an overview of the Dynamic Data Cube technique [3] is given. Since it makes use of the inverse property of some aggregate operators, the problem of summarizing any possible range is reduced to the problem of summarizing and combining ranges that are anchored at $(0, \ldots, 0)$ (like for PS, RPS, SRPS).

The basic DDC technique makes use of non-intersecting boxes which store pre-computed values that only summarize the cells in the box. Those values are stored in the "lower right" surfaces of the box (border cells) and summarize the cells in a region that has the anchor of the box as its anchor and the surface cell as the endpoint. The boxes are organized into a tree that recursively partitions the original data cube. The root node encompasses the entire data cube. It forms children by dividing its range in each dimension in half and storing aggregate values for each box. Each of the children are in turn subdivided into children, and so on. Figure 8 shows an example for a data cube and the corresponding basic DDC cube. Note that the values in the border cells of the same surface are cumulative. Even though empty cells are not stored, the tree consumes more space than the original data cube.

Each of the boxes on the root level contains $(n/2)^d - (n/2-1)^d$ border cells. Since the values in the border cells are cumulative, an update to the anchor cell of a box has to be propagated to all those border cells. To reduce the update cost [3] introduces the $\mathrm{B^c}$ tree. $\mathrm{B^c}$ trees are standard B-trees whose non-leaf nodes are augmented by an auxiliary value that stores the sum of the leafs in the left sub-tree. By taking advantage of these auxiliary values $\mathrm{B^c}$ trees provide balanced query and update costs of $\log m$ for any one-dimensional array of size m that stores cumulative values. By storing the (one-dimensional) border cell arrays in $\mathrm{B^c}$ trees, update and query costs of $\mathrm{O}(\log^2 n)$ can be achieved for two-dimensional data cubes. For data cubes with $d > 2$ dimensions the $(d-1)$-dimensional surfaces that contain the border

[3] The only exception occurs for $k = 1$, when SRPS collapses to the Prefix Sum technique.

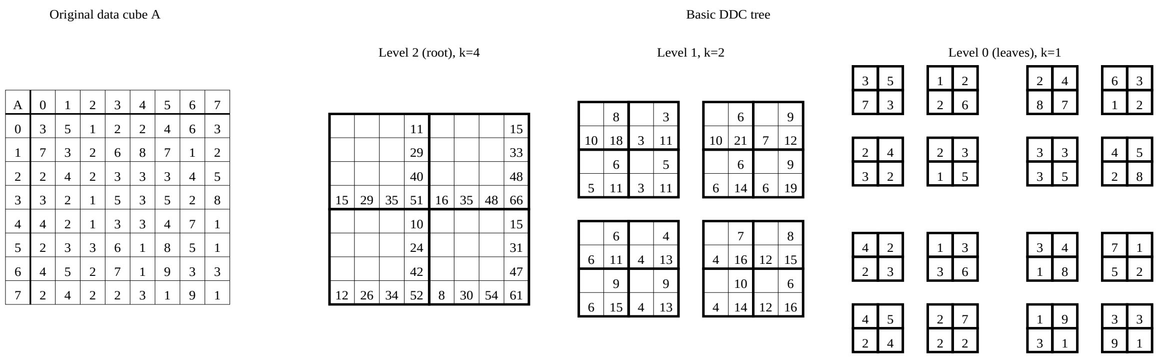


Fig. 8. Original data cube (left) and corresponding basic DDC tree

cells are recursively stored as $(d-1)$-dimensional data cubes. Thus DDC with $\mathrm{B^c}$ trees and the recursive technique for storing the border values guarantees the polylogarithmic update and query costs. Details can be found in [3].

2.4 SDDC: The Space-Efficient Dynamic Data Cube Technique

SDDC balances query and update costs to enable what-if analysis and other applications that require instant fast updates. Like for the Dynamic Data Cube, the costs are $\mathrm{O}(\log^d n)$. SDDC is able to maintain those bounds at much lower storage costs. For clarity, we will first describe a simpler basic approach that has higher update costs, and then SDDC with the polylogarithmic costs.

The Basic SDDC Compared to the Dynamic Data Cube, SDDC uses a different technique to summarize the data. It first partitions the data cube into boxes using the same technique as SRPS, except for two differences. First, the side-length of a box is set to $k = n/2$, i.e., the data cube is partitioned into 2^d boxes of equal size. Second, while the aggregation regions of the border cells remain the same, the inner cells do not store relative prefix sums any more. Instead a recursive approach is taken. More precisely, for each box the region that contains all inner cells of that box (which is a hyper-cube of side-length $n/2 - 1$) is partitioned into 2^d non-intersecting boxes of equal size. Those boxes store the corresponding border cell values relative to the enclosing larger box they partition. Their regions of inner cells are then in turn partitioned into 2^d boxes, and so on. In Figure 9 an example for a data cube and the corresponding basic SDDC cube are shown. Instead of dividing the region of the inner cells in each dimension in half, one could alternatively choose any other partitioning into boxes, where the number of resulting smaller boxes is bounded by a constant. Other partitions, besides the regular binary subdivision, with flexible split positions in the different dimensions can be used to identify similarity regions, which could be exploited by operations that can take advantage of low variance distributions (i.e. for data compression).

Due to how the boxes are created, the complete basic SDDC fits into the space of the original data cube (with n^d cells). The storage savings compared to the basic DDC approach are considerable. The basic DDC creates n^d boxes of side-length 1, $(n/2)^d$ boxes of side-length 2 and so on. In general there are $(n/2^i)^d$ boxes of side-length 2^i and a box of side-length 2^i stores $(2^i)^d - (2^i - 1)^d$ values. To keep the description simple, we assume that $n = 2^j$ for some

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

	0	1	2	3	4	5	6	7	8	9
0	3	5	6	8	10	17	6	9	12	13
1	7	3	2	11	8	33	1	2	7	2
2	9	4	2	3	3	50	4	5	7	4
3	12	9	5	28	14	69	7	15	35	7
4	16	2	1	6	3	86	7	1	11	2
5	18	19	29	54	74	126	25	45	65	77
6	4	5	2	14	1	28	3	3	10	1
7	6	4	2	8	3	42	9	1	13	3
8	11	13	7	30	7	60	13	9	39	9
9	17	1	2	7	2	76	3	1	9	2

Fig. 9. Original data cube (left) and corresponding basic SDDC cube (right)

integer $j \geq 2$. Then the storage requirements of the basic DDC can be expressed as

$$
\begin{aligned}
\sum_{i=0}^{j}(\frac{n}{2^i})^d((2^i)^d - (2^i-1)^d) &= \sum_{i=0}^{j} n^d(1 - (\frac{2^i-1}{2^i})^d) \\
&= n^d \sum_{i=0}^{j}(1 - (\frac{2^i-1}{2^i})^d) \\
&> n^d \sum_{i=0}^{2}(1 - (\frac{2^i-1}{2^i})^d) \\
&= n^d((1 - 0^d) + (1 - (\frac{1}{2})^d) + (1 - (\frac{3}{4})^d)
\end{aligned}
$$

For $d \geq 2$ it holds that $(1/2)^2 \geq (1/2)^d$ and that $(3/4)^2 \geq (3/4)^d$. Applying those inequalities to the above inequality for the storage costs results in

$$
\begin{aligned}
\sum_{i=0}^{j}(\frac{n}{2^i})^d((2^i)^d - (2^i-1)^d) &> n^d(1 + (1 - (\frac{1}{2})^2) + (1 - (\frac{3}{4})^2) \\
&= n^d(3 - \frac{1}{4} - \frac{9}{16}) \\
&> 2n^d
\end{aligned}
$$

The basic DDC therefore required more than twice the space of the original data cube. Thus our new basic SDDC technique reduces the space overhead by more than the size of the original data cube!

The query and update behavior of the basic SDDC is similar to that of the basic DDC. We make the following observation. The boxes of the basic SDDC conceptually form a tree where each node corresponds to a box. To simplify the description, we will use the terms *node* and *box* interchangeably. The root node encompasses the entire data cube which is divided into 2^d boxes. Its children are the nodes that correspond to the regions of inner cells of the boxes the root is partitioned into, and so on. In general, the children of a node are those smaller boxes that partition the region of the inner cells of the node. They store the corresponding border cell values. At the leaf level nodes simply store the value of the single cell that corresponds to the node. Since the side-length of a node is less than half the side-length of its parent, the tree height can not exceed $\log_2 n$. Figure 10 presents the tree that corresponds to the basic SDDC cube from Figure 9.

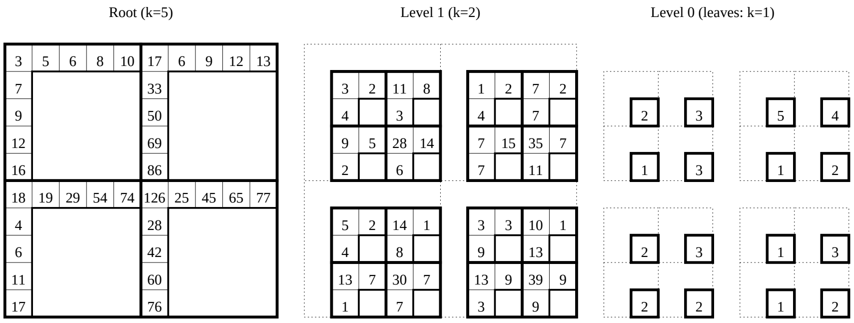


Fig. 10. Example of the basic SDDC tree (empty cells are not stored)

To find the sum for any query region that is anchored at $(0,\ldots,0)$, the tree is descended and the appropriate border values are added. Let q be the endpoint of the query region. The processing of the query starts at the root by finding and accessing that child (i.e., box) that contains cell q. The sum of all values in the relevant cells outside the box and in its border cells is obtained by accessing the appropriate border cells in the same way as this was done for SRPS. The region sum for the relevant inner cells is computed by recursively descending to that child node that contains q, until q falls into a border cell or the accessed node is a leaf. In the example in Figure 11 the cells that are accessed in order to compute $\mathrm{SUM}(A[0,0]:A[8,7])$ are shaded; cell $(8,7)$ is hatched.

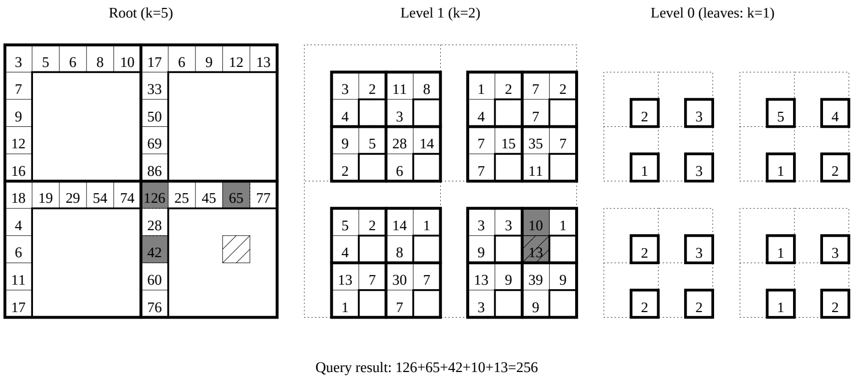


Fig. 11. Cells that contain the relevant data for $\mathrm{SUM}(A[0,0]:A[8,7])$

The partitioning of the data cube is non-overlapping, therefore at most one box per level contains q. At each level the sum of the relevant values outside the box and in the border cells of the box are obtained by accessing at most 2^d-1 border cells (analysis identical to SRPS). Since the tree has at most $\log_2 n$ levels, the query cost for any query anchored at $(0,\ldots,0)$ is less or equal than $(2^d-1)\log_2 n = \mathrm{O}(\log n)$.

Updates on the basic SDDC are very expensive in the worst case. Consider an update to cell $(1,1,\ldots,1)$. At the root the boxes are defined as for the SRPS technique with $k=n/2$. Since there are no prefix sums for inner cells, the update cost at the root node is obtained as $((n/(n/2)+n/2-2)^d)-(n/2-1)^d$ (update cost for corresponding SRPS cube minus update cost for inner cells on that cube). Thus, already at the root $\mathrm{O}(n^{d-1})$ cells have to be updated for a single update on cell $(1,1,\ldots,1)$.

SDDC with Improved Updates The problem the basic SDDC faces regarding updates are similar to the update problem for the basic DDC technique. To reduce the update costs, we can apply the same technique as for DDC, i.e., using B^c trees for balanced update and query costs on two-dimensional data cubes (i.e., one B^c tree for each of the two one-dimensional arrays of border cells), and storing the border values of higher-dimensional data cubes recursively (see Section 2.3). However, B^c trees and the recursive approach introduce unnecessary redundancy.

Recall that the values of the border cells in the same surface of a box are cumulative, which results in the high worst case update costs. B^c trees balance the query and update costs for one-dimensional cumulative arrays. Unfortunately, for an array of size n the corresponding B^c tree contains n leaves and up to $n-1$ internal nodes. Thus B^c trees are not space-efficient. We eliminate the space overhead by using an elegant technique that embeds the tree into the array. The main idea is to first replace the cumulative values by the corresponding differences of the values of neighboring cells (e.g., the array $[3,5,8,16]$ is replaced by $[3,2,3,8]$) and then to apply the basic SDDC technique to this array of differences (in the example this would result in the array $[3,2,(3+2+3),8] = [3,2,8,8]$). Queries and updates are processed as described for the basic SDDC technique, resulting in a worst case cost of $O(\log n)$ for both operations. Thus we have a data structure that replaces the B^c tree and does not add any space overhead compared to storing the original array.

The other source of redundancy is the recursive approach. For instance, for a three-dimensional box the three two-dimensional surfaces that contain the border cells are stored independently as two-dimensional data cubes. Those surfaces have non-empty intersections, namely the cells in the one-dimensional "columns" that contain the anchor of the box. The values of the cells in the intersection are stored for each surface they belong to. This redundancy similarly occurs in higher dimensions. One can simply remove it by storing the values for the cells in the intersections only once. Basically, instead of handling the border surfaces independently and storing multiple copies of cells in intersections, the values are stored in the same overlapping surfaces. This is possible because the new trees that replace the B^c trees fit into the space of the cells they "encode". This makes it possible to embed the values for the recursive computation of the border cell values into the space of exactly those border cells. Consequently SDDC has the *same storage consumption as the original data cube*, thus does not introduce any space overhead.

It can be shown, that our new approach still guarantees a query and update cost of $O(\log^d n)$, which is sublinear in the side-length of the data cube. Details of the analysis are similar to the DDC technique [3] and therefore omitted here. Note that improving on the B^c trees and the recursive technique for storing the values of the border cells further increases the space savings of SDDC compared to DDC.

3 Conclusion

Aggregate range queries are useful tools for analyzing information that is stored in data cubes. For massive data sets, however, accessing and aggregating the relevant data on the fly can result in slow responses that negatively affect the analysis process. In this paper two new techniques were discussed that speed up range queries by storing pre-aggregated information, while still supporting efficient updates. Both greatly improve on previous techniques by providing the same query and update costs, while reducing the space costs by up to or by more than the size of the original data cube, respectively. Using SRPS or SDDC cubes instead of the original data cube thus provides efficient queries and updates without introducing any space overhead.

To be more precise, we developed one technique (SRPS) that guarantees that any aggregate range query is answered in constant time and that no update results in costs higher than the square root of the data cube size. We presented another technique (SDDC), that improved the only existing technique which provides provably polylogarithmic worst case query and update costs for any data cube. Our technique guarantees the same query and update costs, while reducing the space overhead by an amount of space that is greater than the size of the original data cube. This means that if the size of the data cube, i.e. the space required to store the contents of all cells in a multidimensional database, is one gigabyte, then SDDC uses more than one gigabyte *less* than the DDC technique.

Thus our new techniques efficiently support online aggregation for massive data sets. Reducing the space requirements not only saves storage costs, but at the same time reduces the real access and update times. This is not reflected in our cost formulas that are only based on cell accesses. Real access costs, however, also depend on cache sizes and seek/latency times for external storage devices. For real query and update costs, smaller space consumption can be very beneficial. Similar to all methods that are based on prefix sums, our approaches are particularly suited for dense data sets. Our future work will explore techniques for sparse high-dimensional data cubes.

References

1. C.-Y. Chan and Y. E. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. 25th VLDB*, 1999.
2. Transaction Processing Performance Council. TPC-H benchmark (1.1.0). Available at http://www.tpc.org.
3. S. Geffner, D. Agrawal, and A. El Abbadi. The dynamic data cube. In *Proc. EDBT*, 2000. To appear.
4. 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.
5. S. Geffner, M. Riedewald, D. Agrawal, and A. El Abbadi. Data cubes in dynamic environments. *Data Engineering Bulletin*, 22(4), 1999.
6. 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.
7. C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD*, 1997.