The Iterative Data Cube

Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi

Department of Computer Science, University of California, Santa Barbara CA 93106, USA
{mirek, agrawal, amr}@cs.ucsb.edu

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. In this paper a technique is presented that generates a new class of schemes that support range queries on data cubes – Iterative Data Cubes. The main idea is that techniques that provide a certain tradeoff of query and update costs for a one-dimensional data cube are applied iteratively along the dimensions of data cubes of higher dimensionality. The different one-dimensional techniques can be combined, resulting in a great variety of Iterative Data Cubes. We show that Iterative Data Cubes are easy to generate, analyze and implement. They generalize previous seemingly unrelated approaches. Our technique is applicable to aggregate operators and attributes if the attribute's domain forms an abelian group under the aggregate operator (e.g., integers under addition). In this paper we focus on techniques that do not cause any space overhead for dense data cubes.

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 [5]. 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 [5].

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 depend on this cell. 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.

Previous work that deals with this problem is discussed in Section 2. Section 3 summarizes the contribution of this paper. The technique to generate Iterative Data Cubes is described in Section 4. In that section techniques for querying and updating Iterative Data Cubes are discussed as well. Section 5 concludes this article.

2 Previous Work

An elegant algorithm for computing range queries that return the sum of the selected cells in data cubes is presented in [6]. 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. The prefix sum for a certain data cube cell c is the sum of the values of the cells inside a bounding hyper-rectangle which is described by the cell of the data cube with the smallest index in each dimension ("upper left cell" in two-dimensional terminology) and cell c. 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 data cube 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 is done by partitioning the data cube into smaller chunks. In a chunk only some values depend on cells outside the chunk, while most cells contain prefix sums that are local to the chunk. Applying RPS to a data cube, however, leads to a space overhead. This overhead is removed in the *Space-Efficient Relative Prefix Sum* (SRPS) [7] approach which guarantees the same query and update costs as RPS. The *Hierarchical Cubes* techniques [1] generalize the idea of RPS by allowing different tradeoffs between update and query cost. Two different schemes are proposed – Hierarchical Rectangle Cubes (HRC) and Hierarchical Band Cubes (HBC). The tradeoff for both techniques is selected by setting parameters (the hierarchical division of the dimension domains) that control the generation of the pre-computed values. Thus query and update costs depend on those parameters as well as the dimensionality and the domain sizes of the dimensions. Due to the great number of parameter settings and the complexity of the cost functions, it is necessary to experimentally evaluate the different parameter settings in order to find the *best* Hierarchical Cube for an application.

The only known techniques that guarantee that query and update cost are *both* sublinear in the domain size of the dimensions for any data cube are the *Dynamic Data Cube* (DDC) [3] and the *Space-Efficient Dynamic Data Cube* (SDDC) [7]. SDDC improves on DDC by reducing the storage costs while at the same time providing less or equal costs for both queries and updates.

We will further on use the term *pre-aggregated data cube* to refer to a data cube that is obtained from the original data cube by storing pre-aggregated values in some of the cells of the cube (Figure 2).

3 Our Contribution

We present a new technique that generates Iterative Data Cubes (IDC). The motivation for IDC is that it is much easier to develop, analyze and implement techniques that support range queries and updates when the data cube has only *one* dimension, i.e., is basically a

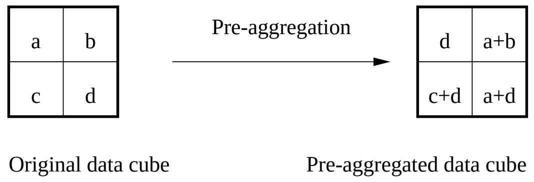

Fig. 2. Original data cube vs. pre-aggregated data cube

one-dimensional array. High-dimensional data cubes are handled by applying to each of its dimensions the one-dimensional technique of choice. Thus a d-dimensional problem is reduced to d one-dimensional problems. We will show that it is straightforward to solve the d-dimensional problem by combining the one-dimensional solutions. To be more precise, we will show that it is an easy task to determine which cells to access and to update for a query or an update, respectively. Deriving formulas for the exact query and update costs is greatly simplified by our technique. Also, Iterative Data Cubes generalize the Prefix Sum, Space-Efficient Relative Prefix Sum[1], Space-Efficient Dynamic Data Cube[2] and Hierarchical Rectangle Cubes.

In this paper we focus on techniques with no space overhead for dense data cubes. The query cost is measured as the number of cells that need to be accessed in order to answer the query. Similarly the update cost is measured in terms of the number of cells of the pre-aggregated data cube whose values have to be updated to reflect an update to any cell of the original data cube. Since dense data cubes are stored and accessed like multidimensional arrays, this cost model is realistic for both, internal (main memory) and external (disk, tape) algorithms.

4 The Iterative Data Cubes Technique

In this section we describe how Iterative Data Cubes are constructed and how queries and updates are processed. The following notation will be used to simplify the description. Let A be a data cube of dimensionality d, and let $c = [c_1, \ldots, c_d]$ be a cell that contains the value $A[c]$.[3] Without loss of generality let the domain of each dimension attribute i be $\{0, 1, \ldots, n_i - 1\}$. $e : f$ 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.

The Iterative Data Cubes technique is described for the aggregate operator SUM. We will refer to range queries that compute the sum of the values of the cells in the selected range as *range sum queries*. Other operators $\oplus$ for which exists an inverse operator $\ominus$ such that for any attribute values a, b it holds that $(a \oplus b) \ominus b = a$ can be handled in a similar way. Popular examples of such operators are COUNT and sum of squares. Note that operators like AVG (average) and standard deviation can also be handled by respectively putting them down to SUM and COUNT, or SUM, sum of squares and COUNT. This, of course, accordingly increases the storage requirements.

[1] The Relative Prefix Sum is made obsolete by the Space-Efficient Relative Prefix Sum technique.
[2] The Dynamic Data Cube is made obsolete by the Space-Efficient Dynamic Data Cube technique.
[3] 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).

4.1 The One-Dimensional Case

Let A be a one-dimensional data cube, i.e., an array with n elements $A[0], A[1], \ldots, A[n-1]$. Storing A requires n cells, i.e., leads to a storage cost of n.[4] An update to any cell has a cost of 1, while the cost of a range sum query is n in the worst case (for $\mathrm{SUM}(A[0] : A[n-1])$). Alternatively the corresponding prefix sum array P can be stored. This array has n elements $P[0], P[1], \ldots, P[n-1]$, such that $P[i] = \sum_{j=0}^{i} A[j]$ $(0 \le i \le n-1)$, thus also requiring n cells of storage, i.e., not introducing any space overhead. Any range sum query on A can be answered by accessing at most two cells of array P. More precisely, for any $0 \le i \le j \le n-1$ the value of $\mathrm{SUM}(A[i] : A[j])$ is equal to $P[j] - P[i-1]$ (for convenience define $P[-1] = 0$). Updates to a single cell of the original array, however, affect all cells in P with greater or equal index values, resulting in a worst case update cost of n. In Figure 3 an example array and the corresponding prefix array are shown. The query range and the updated cell are framed, the corresponding cells that are accessed are shaded.

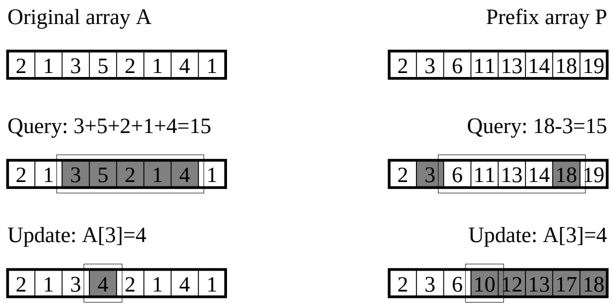

Fig. 3. Original array A and corresponding prefix array P (query range and updated cell are framed, accessed cells are shaded)

The two techniques of storing A directly or storing its prefix sum array P instead constitute the extreme cases of a maximal support for updates and a maximal support for queries, respectively. In a (worst case query cost, worst case update cost)-graph they define the points $(n, 1)$ and $(2, n)$. Figure 4 shows a graphical representation; below techniques are presented that define the other points in the graph. Note, that tradeoff $(1, n)$ can not be reached. A technique that guarantees that any range sum query can be answered by accessing at most one cell needs to pre-compute and store the answers for all possible range sum queries. For $n > 2$, however, more than n range sums depend on $A[1]$ (cell [1] is included in $2n-2$ ranges), i.e., updating this value results in costs greater than n.

Between the two extremes other (worst case query cost, worst case update cost) tradeoffs are possible. Selected techniques are presented below. For clarity, techniques that partition the array into smaller chunks are described for the case that the array can be partitioned evenly, i.e., without leaving "small fractions". The cost formulas, however, are valid in general. Due to space limitations we do not provide details about how known techniques are used to construct the corresponding pre-aggregated arrays and how queries and updates are processed. Those details can be found in the cited papers.

One-Dimensional SRPS The SRPS array for an array A is constructed by first partitioning A into k chunks of equal size. The anchor of a chunk (its leftmost cell) contains the sum of

[4] Special cases like sparse arrays are not discussed here.

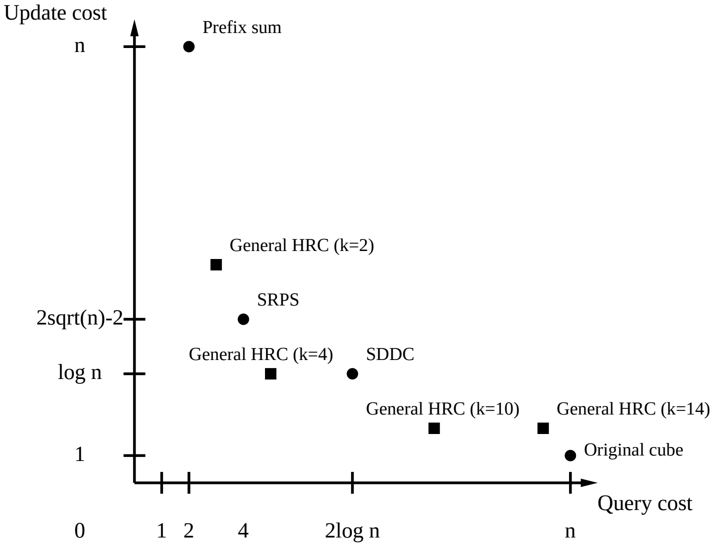

Fig. 4. Selected tradeoffs for the worst case query and update cost for a one-dimensional array of size $n = 16$ (relative position of points can be different for other n)

the values of all cells in A with a less or equal index. The other cells of the chunk contain the prefix sum relative to the right neighbor of the anchor. A query $\mathrm{SUM}(A[0] : A[c])$ is answered by adding $P[c]$ and the value stored in the anchor cell of the chunk that contains c (if different from c). A general range query $\mathrm{SUM}(A[l] : A[u])$ for $l > 0$ is answered as $\mathrm{SUM}(A[0] : A[u]) - \mathrm{SUM}(A[0] : A[l-1])$. Consequently any range sum query can be answered by accessing at most 4 cells in P. An update to a cell that is an anchor of a chunk only propagates to anchors of chunks with a greater or equal index. Updates to any other cell additionally affect all cells with a greater or equal index in the same chunk. Thus the worst case query and update costs are 4 and $n/k+k-2$ [7], respectively. The update cost is minimized when $k = \lfloor\sqrt{n}\rfloor$ or $k = \lceil\sqrt{n}\rceil$. If n is a perfect square then the optimal update cost simplifies to $2\sqrt{n} - 2$. Figure 5 shows an example for an SRPS array that was obtained for a chunk size of $k = 5$ and how $\mathrm{SUM}(A[2] : A[6])$ and and update to $A[3]$ are processed.

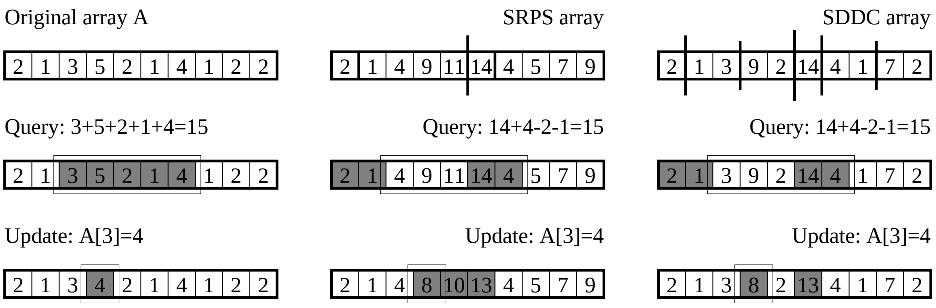

Fig. 5. Original array A and corresponding SRPS and SDDC arrays (query range and updated cell are framed, accessed cells are shaded)

One-Dimensional SDDC Similarly we can apply the SDDC technique to a one-dimensional array without incurring any space overhead. To compute P, the original array A is first partitioned into two halves. The anchor of the left half, [0], contains the value $A[0]$. The anchor of the right half, $[n/2]$, contains $\mathrm{SUM}(A[0] : A[n/2])$. To the two arrays of inner cells $[1] : [n/2-1]$ and $[n/2+1] : [n-1]$ the same technique is applied recursively. The recursive partitioning defines a hierarchy, more precisely a tree of height less or equal to $\lceil \log_2 n \rceil$, on the partitions (chunks). Queries and updates conceptually descend this tree. The processing starts at the root and continues to that chunk that contains the endpoint of the query or the updated cell, respectively. A query $\mathrm{SUM}(A[0] : A[c])$ is answered by adding the values of the anchors of those chunks that contain c. Due to the construction at most one chunk per level can contain c, resulting in a worst case query cost of $\lceil \log_2 n \rceil$. Queries with general ranges are reduced to two queries of the form $\mathrm{SUM}(A[0] : A[c])$ (as described before for SRPS). Thus the cost of answering any range query is bounded by $2\lceil \log_2 n \rceil$. At each level an update to a cell c in a chunk C only propagates to those cells that have a greater or equal index than c and are an anchor of a chunk that has the same parent as C. Consequently the update cost is bounded by the height of the tree, i.e. $\lceil \log_2 n \rceil$. In Figure 5 an example of a SDDC array and how the query $\mathrm{SUM}(A[2] : A[6])$ and an update to $A[3]$ are processed are shown.

Note that a simple generalization of this technique can generate (worst case query cost, worst case update cost) tradeoffs of $(2\lceil \log_k n \rceil, (k-1)\lceil \log_k n \rceil)$ when the array is recursively partitioned into k chunks of equal size. We restrict k to $k > n/2$, since otherwise the technique collapses to the Prefix Sum. Note that for $k = \sqrt{n}$ the same cost tradeoff is obtained as for one-dimensional SRPS.

A slight modification of the SDDC technique creates another family of tradeoffs. Let the partitioning of the array be performed as described for SDDC. But instead of storing relative prefix sums in the anchor cells of a chunk, the sum of the values in the chunk is stored in its endpoint. Intuitively the pre-aggregated values are local to their chunks. Queries and updates are processed similarly as for SDDC. The main difference is that now queries have to access up to $2(k-1)$ cells per level, while updates affect at most one value per level. Consequently the query and update costs are bounded by $2(k-1)\lceil \log_k n \rceil$ and $\lceil \log_k n \rceil$, respectively. We will refer to this technique as SDDC2.

Generalized One-Dimensional HRC For an array A of length n the corresponding Hierarchical Rectangle Cube P is generated by first partitioning the array into k_1 chunks of equal size. Those chunks form the top level (level 1), their anchors (leftmost cells) are denoted as *root* cells. On the level below (level 2) each of the chunks is partitioned into k_2 smaller chunks of equal size. This continues recursively until level l is reached such that $\prod_{i=1}^{l-1} k_i = n$. Note that each of the k_i is an integer such that $1 \leq k_i \leq n$. The *parent* of a cell c in chunk C at level $i > 1$ is defined as the anchor of that chunk C' at level $i-1$ that is the ancestor of C (i.e., chunk C is one of the k_{i-1} partitions of C'). A root cell of the HRC cube P contains the value of the corresponding cell in the original array A. Any other cell c in P contains the value $\mathrm{SUM}(A[\mathrm{parent}(c)] : A[c])$ (Figure 6). Consequently the cells of a one-dimensional Hierarchical Rectangle Cube contain the prefix sum relative to their respective parents.

We generalize the Hierarchical Rectangle Cube scheme to GHRC as follows. Array A is partitioned into k chunks of size $s_1, s_2, \ldots, s_k$, respectively. Any cell in the pre-aggregated array P contains the prefix sum relative to the anchor of the chunk it is contained in. While the hierarchical partitioning scheme of the HRC technique restricts the choice of k and $s_1, \ldots, s_k$, the generalization allows for any $1 \leq k \leq n$ and $1 \leq s_i \leq n$ ($i \in \{1, 2, \ldots, k\}$) such that $\sum_{i=1}^{k} s_i = n$. A range query is answered by adding the values of all chunk endpoints that are contained in the query range, adding the value of the cell at the endpoint of the query range (if it

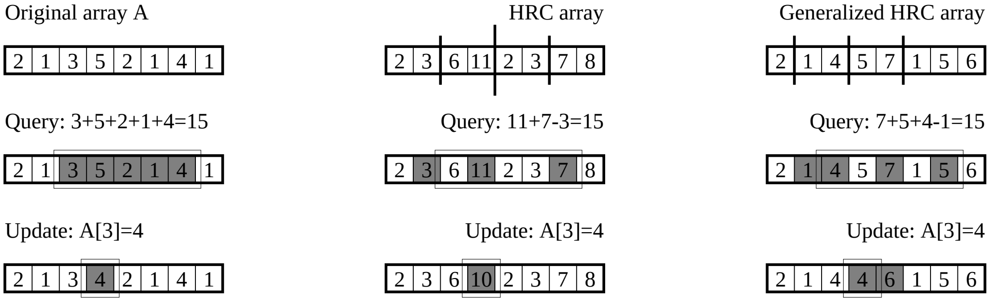

Fig. 6. Original array A and corresponding HRC and GHRC arrays (query range and updated cell are framed, accessed cells are shaded)

is not an endpoint of a chunk) and subtracting the value of the cell left to the anchor of the query range (if it is not an endpoint of a chunk). Thus the query cost is bounded by $k+1$. Figure 6 shows an example. Updates only affect cells in the same chunk that have a greater or equal index than the updated cell, resulting in a worst case update cost of $\max\{s_1, \ldots, s_k\}$. For a certain k the query cost is fixed, but the update cost is minimized by choosing $s_1 = s_2 = \ldots = s_k$. The corresponding family of (worst case query cost, worst case update cost) tradeoffs thus becomes $(k+1, \lceil n/k \rceil)$.

4.2 Constructing Iterative Data Cubes with more than One Dimension

For high-dimensional data cubes one-dimensional techniques are applied along each dimension. Here we assume that only techniques are used that do not have any space overhead. Let (q_i, u_i) be the query-update cost tradeoff of the one-dimensional technique selected for dimension δ_i. Table 7 summarizes the different one-dimensional techniques and their query-update cost tradeoffs. To keep the following description simple we will slightly abuse the notation and use (q_i, u_i) also to refer to the technique with the corresponding query-update cost tradeoff.

One-dimensional technique	**Query cost** (worst case)	**Update cost** (worst case)	**Note**
Original array	n	1	
Prefix Sum (PS)	2	n	
Space-Efficient Relative Prefix Sum (SRPS)	4 4	$2\sqrt{n}-2$ $n/k+k-2$	when n perfect square in general for $2 \le k < n$
Space-Efficient Dynamic Data Cube (SDDC)	$2\lceil \log_2 n \rceil$	$\lceil \log_2 n \rceil$	
Generalized SDDC	$2\lceil \log_k n \rceil$	$(k-1)\lceil \log_k n \rceil$	$2 \le k \le n/2$
SDDC2	$2(k-1)\lceil \log_k n \rceil$	$\lceil \log_k n \rceil$	$2 \le k \le n/2$
Generalized Hierarchical Rectangle Cube (GHRC)	$k+1$	$\lceil n/k \rceil$	$2 \le k < n$

Fig. 7. Query-update cost tradeoffs for selected one-dimensional techniques

The algorithm first applies technique (q_1, u_1) to dimension δ_1. To be more precise, (q_1, u_1) is applied to each array $[0, i_2, i_3, \ldots, i_d] : [n_1 - 1, i_2, i_3, \ldots, i_d]$ (for any combination of i_j where $0 \le i_j < n_j$ for $j \in \{2, 3, \ldots, d\}$). Let the resulting pre-aggregated data cube be A_1. Clearly A_1 does not contain more cells than A (since (q_1, u_1) does not use additional space) and can be computed at a cost of $n_2 \cdot n_3 \cdots n_d \cdot C_1(n_1)$ where $C_1(n_1)$ denotes the cost of applying technique (q_1, u_1) to an array of size n_1.

In the next step technique (q_2, u_2) is similarly applied to dimension δ_2, but now with A_1, the result of the previous step, as the input data cube. This process continues until all dimensions are processed. The cost for processing dimension δ_j is $C_j(n_j)/n_j \cdot \prod_{i=1}^{d} n_i$. This results in a total construction cost of $\prod_{i=1}^{d} n_i \cdot (\sum_{j=1}^{d} C_j(n_j)/n_j)$ which simplifies to $n^{d-1} \sum_{j=1}^{d} C_j(n)$ for data cubes with dimensions that have the same domain size n. Figure 8 shows an example for a two-dimensional data cube where the one-dimensional SRPS technique is applied along both dimensions.

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

Original data cube

A1	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	2	4	6	11	3	6	21	5	12
3	3	2	3	11	3	8	21	8	10
4	4	2	3	10	3	7	24	1	4
5	2	3	6	14	1	9	28	1	2
6	4	5	7	18	1	10	31	3	7
7	2	4	6	10	3	4	23	1	4
8	5	4	7	13	3	5	19	9	15

Processing of the rows using SRPS

P	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

Processing of the columns using SRPS

Fig. 8. Original data cube A, intermediate cube A_1, and final SRPS cube $P(= A_2)$

Querying and Updating Iterative Data Cubes A range query on the original one-dimensional data cube A has to access all cells in the selected range. On the corresponding Iterative Data Cube P the range translates to a set of cells which depends on the pre-aggregation technique used to generate P. Similarly, while the update of a single data item only affects one cell in A, it will affect a set of cells in P. For instance, when using the one-dimensional SRPS technique for $n = 10$, then SUM$(A[2] : A[6])$ can be computed as $P[5] + P[6] - P[0] - P[1]$, i.e., range $[2] : [6]$ translates to $\{0, 1, 5, 6\}$ (see SRPS array in Figure 5). Similarly an update to $A[3]$ affects $P[3]$, $P[4]$ and $P[5]$, i.e. $[2]$ translates to $\{3, 4, 5\}$.

For data cubes with more than one dimension the affected cells of the Iterative Data Cube P are obtained by combining the index sets for each dimension. For a query SUM$(A[l_1, l_2, \ldots, l_d] : A[u_1, u_2, \ldots, u_d])$ on a d-dimensional data cube let Q_i be the index set for dimension δ_i. Q_i is obtained by translating the query range $[l_i] : [u_i]$ to the corresponding index set. Note that it has to be distinguished between cells whose values are added and cells whose values are subtracted. This is done by marking indexes of cells whose values are to be subtracted with a "$-$" (minus). For each dimension δ_i a set Q_i is generated independently. The set of cells c in P that have to be accessed in order to answer the query is then defined as $\{c = [c_1, c_2, \ldots, c_d] | c_i \in Q_i \text{ for all } 1 \leq i \leq d\}$. The value of a cell c from this set is added if the number of indexes c_i $(1 \leq i \leq d)$ which are marked with a "$-$" is even, it is subtracted otherwise. In Figure 9 an example is shown for a two-dimensional SRPS data cube. The range $2 : 5$ in the first dimension is translated into the index set $\{0, 1, 3, 5\}$ where 0 and 1 are marked with a "$-$". Similarly the set $\{3, 6\}$ (3 marked with "$-$") is obtained for the range $4 : 6$ in the second dimension. The combination of the two index sets results in the set $\{[0, 3], [0, 6], [1, 3], [1, 6], [3, 3], [3, 6], [5, 3], [5, 6]\}$. Since 0 from the first index set and 3 from the second one are both marked with "$-$", the value $P[0, 3]$ is added to the result. On the other hand, 6 from the second index set is not marked, therefore

$P[0,6]$ has to be subtracted. The set of cells of P that are affected by an update to a cell of the original data cube can be computed similarly.

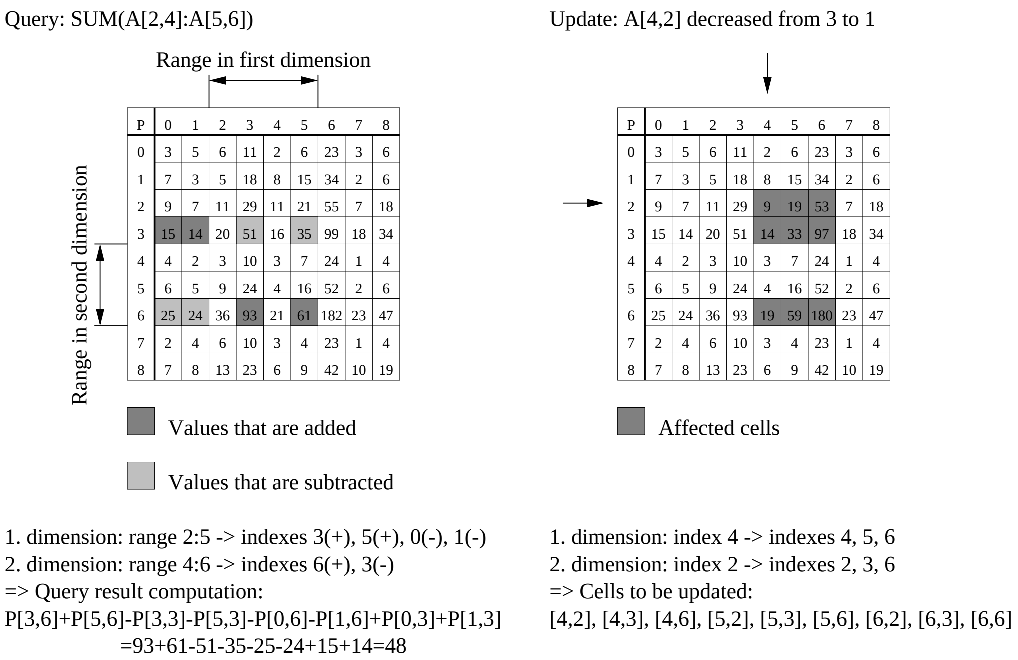

Fig. 9. Processing queries and updates on an Iterative Data Cube (SRPS technique used for both dimensions)

The iterative construction technique basically reduces the d-dimensional problem to d one-dimensional problems. Once the query and update cost for a one-dimensional technique is known, the costs for a d-dimensional data cube are obtained by simply multiplying the costs for the single dimensions. Since it is a comparably easy task to analyze worst case and average costs for one-dimensional data cubes, it is as easy to analyze the costs for any d-dimensional Iterative Data Cube. For example, let A be a three-dimensional data cube with $n_1 = n_2 = n_3 = n$. Techniques $(2\log_2 n, \log_2 n)$, $(n, 1)$ and $(4, 2\sqrt{n} - 2)$ are applied along dimensions δ_1, δ_2 and δ_3, respectively. Then the corresponding Iterative Data Cube has a worst case query cost of $(2\log_2 n)\cdot n\cdot 4 = 8n\log_2 n$ and a worst case update cost of $\log_2 n\cdot 1\cdot(2\sqrt{n}-2) = 2(\sqrt{n}-1)\log_2 n$.

An interesting aspect is that it is not only possible to build an Iterative Data Cube that has the same query-update cost tradeoff as previous schemes like the PS, SRPS, SDDC and HRC. Applying those techniques iteratively along each dimension also leads to the same final pre-aggregated data cube as applying the d-dimensional technique directly. Thus each of the mentioned schemes can be analyzed by analyzing the corresponding Iterative Data Cube instead. This dramatically simplifies the analysis process for those techniques.

5 Conclusion

Iterative Data Cubes efficiently support range queries and updates on data cubes at no storage overhead. There exists a great number of one-dimensional aggregation techniques and it is possible to apply a different technique along each dimension of the data cube. Thus a great variety of Iterative Data Cubes can be constructed – ranging from cubes that are biased towards fast queries to cubes whose low update cost allows them to be used in very dynamic environments while still supporting fast range queries.

The iterative construction technique is easy to implement, and Iterative Data Cubes are easy to analyze. Once the characteristics of an aggregation technique are understood for the one-dimensional case, analyzing a higher-dimensional data cube only requires combining the independently obtained one-dimensional results. For example, the worst case update and query costs of a d-dimensional Iterative Data Cube equal the product of the worst case costs of the d used one-dimensional techniques. Since Iterative Data Cubes generalize some of the earlier pre-aggregation techniques, they also provide a new and simpler way of analyzing and understanding those earlier approaches.

The presented technique is particularly suited for dense data cubes. Certain aggregate operators like MEDIAN, MIN and MAX are not supported by our technique. We are therefore currently examining if the Iterative Data Cube technique can be extended to sparse data sets and other operators.

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. 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.
6. C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD*, 1997.
7. M. Riedewald, D. Agrawal, A. El Abbadi, and R. Pajarola. Space-efficient data cubes for dynamic environments. Technical Report TRCS00-05, UC Santa Barbara, 2000.