Flexible Data Cubes for Online Aggregation

Mirek Riedewald, Divyakant Agrawal, and Amr El Abbadi

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

Abstract. Applications like Online Analytical Processing depend heavily on the ability to quickly summarize large amounts of information. Techniques were proposed recently that speed up aggregate range queries on MOLAP data cubes by storing pre-computed aggregates. These approaches try to handle data cubes of any dimensionality by dealing with all dimensions at the same time and treat the different dimensions uniformly. The algorithms are typically complex, and it is difficult to prove their correctness and to analyze their performance. We present a new technique to generate Iterative Data Cubes (IDC) that addresses these problems. The proposed approach provides a modular framework for combining one-dimensional aggregation techniques to create space-optimal high-dimensional data cubes. A large variety of cost tradeoffs for high-dimensional IDC can be generated, making it easy to find the right configuration based on the application requirements.

1 Introduction

Data cubes are used in Online Analytical Processing (OLAP) [4] to support the interactive analysis of large data sets, e.g., as stored in data warehouses. Consider a data set where each data item has d *functional attributes* and a *measure attribute*. The functional attributes constitute the dimensions of a d-dimensional hyper-rectangle, the data cube. A *cell* of the data cube is defined by a unique combination of dimension values and stores the corresponding value of the measure attribute. An example of a data cube defined for a view on the TPC-H benchmark database [18] might have the total price of an order as the measure attribute and the region of a customer and the order date as the dimensions. It 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 do not need to access and join the "raw" data in the different tables. Instead the information is readily available and hence can be aggregated and summarized from the data cube. Our work focuses on Multidimensional OLAP (MOLAP) systems [14] where data cubes are represented in terms of multidimensional arrays (e.g., dense data cubes).

An *aggregate range query* selects a hyper-rectangular region of the data cube and computes the aggregate of the values of the cells in this region. For interactive analysis it is mandatory to provide fast replies for these queries, no matter how large the selected region. To achieve this, aggregate values for regions of the data cube are pre-computed and stored to reduce on-the-fly aggregation costs. We will refer to a data cube that contains such pre-computed values as a *pre-aggregated* data cube. Whenever necessary, the term *original* data cube is used for a cube without such pre-computed aggregates (i.e., which is obtained directly from the data set). Note, that pre-computation increases update costs since an update to a single cell of the original data cube has to be propagated to all cells in the pre-computed data cube that depend on the updated value. Also, storing *additional* values increases the storage cost. The choice of the query-update-storage cost tradeoff depends on the application. While "what-if" scenarios and stock trading applications require fast updates, for other applications overnight batch processing of updates suffices. But even batch processing poses limits on the update cost which depend on the frequency of updates and the tolerated period of inaccessibility of the data.

In this paper space-optimal techniques for MOLAP systems are explored, i.e., Iterative Data Cubes are generated by *replacing* values of the original data cube with pre-computed aggregates. The space-optimality argument would not apply to sparse data cubes where empty cells are not stored (e.g., Relational OLAP [14]). The main contributions of Iterative Data Cubes are:

[⋆] This work was partially supported by NSF grants EIA-9818320, IIS-98-17432, and IIS-99-70700.

1. For each dimension a different one-dimensional technique for pre-computing aggregate values can be selected. Thus specific properties of a dimension, e.g., hierarchies and domain sizes, can be taken into account.
2. Combining the one-dimensional techniques is easy. This greatly simplifies developing, implementing and analyzing IDCs. In contrast to previous approaches, dealing with a high-dimensional IDC is as simple as dealing with the one-dimensional case.
3. IDCs offer a greater variety of cost tradeoffs between queries and updates than any previous technique and cause no space overhead.
4. They generalize some of the previous approaches, thus providing a new framework for comparing and analyzing them. For the other known techniques we show analytically that our approach at least matches their query-update performance tradeoffs.

In Sect. 2 related work is presented. The Iterative Data Cube technique is described in Sect. 3. There algorithms for querying and updating Iterative Data Cubes are discussed as well. Section 4 contains examples for one-dimensional pre-aggregation techniques and illustrates how those techniques can be used for an application. In Sect. 5 we discuss how IDC performs compared to the previous approaches. Section 6 concludes this paper.

2 Related Work

An elegant algorithm for pre-aggregation on MOLAP data cubes is presented in [11]. We refer to it as the *Prefix Sum* technique (PS). The essential idea is to store pre-computed aggregate information so that range queries are answered in constant time (i.e., independent of the selected ranges). This kind of pre-aggregation results in high update costs. In the worst case, an update to a single cell of the original data cube requires recomputing the whole PS cube. The *Relative Prefix Sum* technique (RPS) [6] reduces the high update costs of PS, while still guaranteeing a constant query cost. RPS is improved by the *Space-Efficient Relative Prefix Sum* (SRPS) [16] which guarantees the same query and update costs as RPS, but uses less space. For dynamic environments Geffner et al. proposed the *Dynamic Data Cube* (DDC) [5] which balances query and update costs such, that both are provably poly-logarithmic in the domain size of the dimensions for any data cube. DDC causes a space overhead which is removed by the *Space-Efficient Dynamic Data Cube* (SDDC) [16]. SDDC improves on DDC by reducing the storage costs, while at the same time providing less or equal costs for both queries and updates. The *Hierarchical Cubes* techniques (HC) [3] generalize the idea of RPS and SRPS by allowing different tradeoffs between update and query cost. Two different schemes are proposed – Hierarchical Rectangle Cubes (HRC) and Hierarchical Band Cubes (HBC).

The above techniques are the ones that are most related to IDC. They explore query-update cost tradeoffs at no extra storage space (except RPS and DDC, which were replaced with the space-efficient SRPS and SDDC) for MOLAP data cubes. Like IDC they are only applicable when the aggregate operator is invertible (e.g., `SUM`) or can be expressed with invertible operators (e.g., `AVG` (average)). Iterative Data Cubes generalize PS, SRPS, and SDDC. For Hierarchical Cubes we show that no better query-update cost tradeoffs than for IDC can be obtained. Note that all of the above techniques, except PS, are difficult to analyze when the data cube has more than one dimension. For instance, the cost formulas for the Hierarchical Cubes are so complex, that they have to be evaluated experimentally in order to find the "best suited" HC for an application.

In [7] a new SQL operator, `CUBE` or "data cube", was proposed to support online aggregation by pre-computing query results for queries that involve grouping operations (`GROUP BY`). Our notion of a data cube is slightly different from the terminology in [7]. More precisely, the *cuboids* generated by `CUBE` (i.e, the results of grouping the data by subsets of the dimensions) are data cubes as defined in this paper. The introduction of the `CUBE` operator [7] generated a significant level of interest in techniques for efficient computation and support of this operator [1, 2, 8, 9, 10, 12, 13]. These techniques do not concentrate on efficient range queries, but rather on which cuboids to pre-compute and how to efficiently access them (e.g., using index structures). Since our technique can be applied to any cuboid which is dense enough to be stored as a multidimensional array, IDC complements research regarding the `CUBE` operator. For instance, by adapting the formulas for query and update costs, support for range queries can be included into the framework for selecting "optimal" cuboids to be materialized. The fact that Iterative Data Cubes are easy to analyze greatly simplifies this process.

Smith et al. [17] develop a framework for decomposing the result of the `CUBE` operator into view elements. Based on that framework algorithms are developed that for a given population of queries select the optimal non-redundant set of view elements that minimizes the query cost. An Iterative Data Cube has properties similar to a non-redundant set of view elements. It contains aggregates for regions of the original data cube, does not introduce space overhead, and allows the reconstruction of the values of the cells of the original data cube. However, in contrast to [17] the goal of IDC is to support *all* possible range queries in order to provide provably good worst case or average query and update costs.

Vitter et al. [19, 20] propose approximating data cubes using the wavelet transform. While [19] explicitly deals with the aspect of sparseness (which is not addressed in this paper) [20], like IDC, targets MOLAP data cubes. Wavelets offer a compact representation of the data cube on multiple levels of resolution. This makes them particularly suited for returning fast approximate answers. Using wavelets to encode the original data cube, however, increases the update costs and does not result in a better worst case performance when exact results are required. While [20] proposes encoding the pre-aggregated data cube which is used for the PS technique, any pre-aggregated (or the original) data cube can be encoded using wavelets. In that sense wavelet transform and IDC are orthogonal techniques[1]. Once an appropriate Iterative Data Cube is selected, approximate answers to queries can be supported by encoding this IDC using wavelet transform.

3 The Iterative Data Cubes Technique

In this paper we focus on techniques for MOLAP data cubes that are handled similar to multidimensional arrays. The query cost is measured in terms of the number of cells that need to be accessed in order to answer the query. Similarly the update cost is measured as the number of cells of the pre-aggregated data cube whose values must be updated to reflect a single update on the data set. Since the data cubes are stored and accessed using multidimensional arrays, this cost model is realistic for both, internal (main memory) and external (disk, tape) algorithms.

In general the IDC technique can be applied to an attribute whose domain forms an Abelian group under the aggregate operator. Stated differently, it can be applied to an aggregate operator $\oplus$ if there exists an inverse operator $\ominus$, such that for all attribute values a and b it holds that $(a \oplus b) \ominus b = a$ (e.g., `COUNT`, but also `AVG` when expressed with "invertible" operators `SUM` and `COUNT`). For the sake of simplicity, the technique is described for the aggregate operator `SUM` and a measure attribute whose domain is the set of integers.

3.1 Notation

Let A be a data cube of dimensionality d, and let without loss of generality the domain of each dimension attribute δ_i be $\{0, 1, \ldots, n_i - 1\}$. A cell $c = [c_1, \ldots, c_d]$, where each c_i is an element of the domain of the corresponding dimension, contains the measure value $A[c]$. With $e : f$ we denote 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. The anchor and endpoint of the entire data cube are $[0, \ldots, 0]$ and $[n_1 - 1, \ldots, n_d - 1]$, respectively. The term $\texttt{op}(A[e] : A[f])$ denotes the result of applying the aggregate operator `op` to the values in region $e : f$. Consequently, $\texttt{SUM}(A[e] : A[f])$ is a *range sum.* The range sum $\texttt{SUM}(A[0, \ldots, 0] : A[f])$ will be referred to as a *prefix sum.*

3.2 Creating Iterative Data Cubes

Iterative Data Cubes are constructed by applying one-dimensional pre-aggregation techniques along the dimensions. To illustrate this process, it is first described for one-dimensional data cubes and then generalized. Let Θ be a one-dimensional pre-aggregation technique and A be the original one-dimensional

[1] Note, however, that wavelet encoding typically increases the update cost.

data cube with n cells. Technique Θ generates a pre-aggregated array A_Θ of size n, such that each cell of A_Θ stores a *linear combination* of the cells of A:

$$\forall 0 \leq j \leq n-1: \quad A_\Theta[j] = \sum_{k=0}^{n-1} \alpha_{j,k} A[k] \ . \tag{1}$$

The variables $\alpha_{j,k}$ are real numbers that are determined by the pre-aggregation technique. Figure 1 shows an example. The array SRPS is the result of applying the SRPS technique with block size 3 to the original array A. SRPS pre-aggregates a one-dimensional array as follows. A is partitioned into blocks of equal size. The anchor of a block $a : e$ (its leftmost cell) contains the corresponding prefix sum of A, i.e., SRPS$[a] = \texttt{SUM}(A[0] : A[a])$. Any other cell c of the block stores the "local prefix sum" SRPS$[c] = \texttt{SUM}(A[a+1] : A[c])$. Consequently, the coefficients in the example are $\alpha_{0,0} = 1$, $\alpha_{1,1} = 1$, $\alpha_{2,1} = \alpha_{2,2} = 1$, $\alpha_{3,k} = 1$ for $0 \leq k \leq 3$, $\alpha_{4,4} = 1$, $\alpha_{5,4} = \alpha_{5,5} = 1$, $\alpha_{6,k} = 1$ for $0 \leq k \leq 6$, and $\alpha_{j,k} = 0$ for all other combinations of j and k.

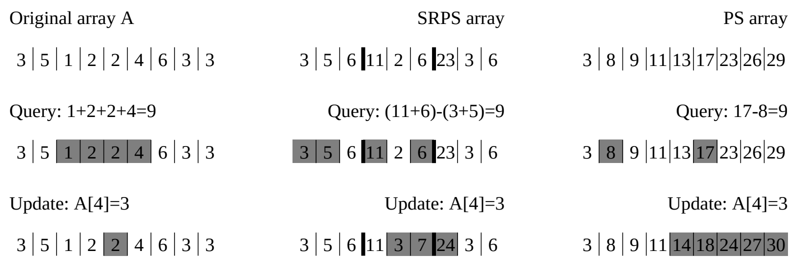
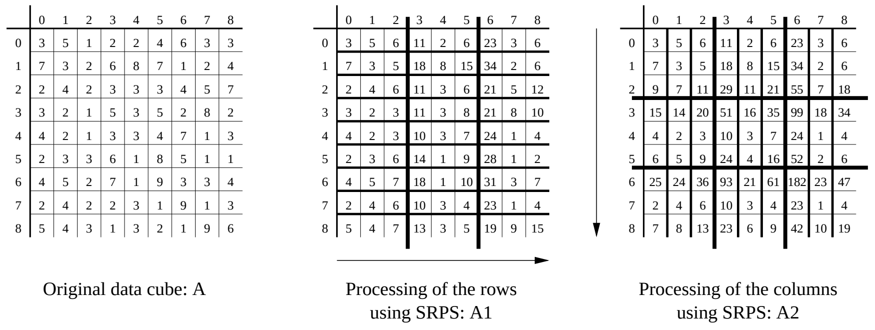


Fig. 1. Original array A and corresponding SRPS (block size 3) and PS arrays (query range and updated cell are framed, accessed cells are shaded)

For a two-dimensional data cube A two (possibly different) one-dimensional pre-aggregation techniques Θ_1 and Θ_2 are selected. Θ_1 is first applied along dimension δ_1, i.e., each row of A is pre-aggregated as described above. Let A_1 denote the resulting pre-aggregated data cube. The columns of A_1 are then processed using technique Θ_2, returning the final pre-aggregated data cube A_2. Figure 2 shows an example. For both dimensions the SRPS technique with block size 3 was selected. Note that applying the two-dimensional SRPS technique directly would generate the same pre-aggregated data cube.



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

Processing of the rows using SRPS: 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 columns using SRPS: A2

	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. 2. Original data cube A, intermediate cube A_1, and final SRPS cube A_2 (fat lines indicate partitioning into blocks by SRPS)

Generalizing the two-dimensional IDC construction to d dimensions is straightforward. First, for each dimension δ_i, $1 \leq i \leq d$, a one-dimensional technique Θ_i is selected. Then Θ_1 is applied along dimension δ_1, i.e., to each array $[0, c_2, c_3, \ldots, c_d] : [n_1 - 1, c_2, c_3, \ldots, c_d]$ for any combination of c_j,

$0 \leq c_j < n_j$ and $j \in \{2, 3, \ldots, d\}$ (intuitively only the first dimension value varies, while the others are fixed). Let the resulting pre-aggregated data cube be A_1. Each cell $c = [c_1, \ldots, c_d]$ in A_1 now contains a linear combination of the values in the original array A which are in the same "row" along δ_1. Formally,

$$A_1[c_1, c_2, \ldots, c_d] = \sum_{k_1=0}^{n_1-1} \alpha_{1,c_1,k_1} A[k_1, c_2, \ldots, c_d] \ . \tag{2}$$

Clearly A_1 does not contain more cells than A (since Θ_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 $C_i(n_i)$ denotes the cost of applying technique Θ_i to an array of size n_i. In the next step technique Θ_2 is similarly applied to dimension δ_2, but now with A_1, the result of the previous step, as the input data cube. For all cells in the resulting cube A_2 it holds that

$$\begin{aligned} A_2[c_1, c_2, \ldots, c_d] &= \sum_{k_2=0}^{n_2-1} \alpha_{2,c_2,k_2} A_1[c_1, k_2, c_3, \ldots, c_d] & (3)\\ &= \sum_{k_2=0}^{n_2-1} \alpha_{2,c_2,k_2} \sum_{k_1=0}^{n_1-1} \alpha_{1,c_1,k_1} A[k_1, k_2, c_3, \ldots, c_d] & (4)\\ &= \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \alpha_{1,c_1,k_1} \alpha_{2,c_2,k_2} A[k_1, k_2, c_3, \ldots, c_d] \ . & (5) \end{aligned}$$

This process continues until all dimensions are processed. The final result, the pre-aggregated data cube A_d, contains values which are the linear combination of the values in the original data cube. More precisely

$$A_d[c_1, c_2, \ldots, c_d] = \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \cdots \sum_{k_d=0}^{n_d-1} \alpha_{1,c_1,k_1} \alpha_{2,c_2,k_2} \cdots \alpha_{d,c_d,k_d} A[k_1, k_2, \ldots, k_d] \ . \tag{6}$$

The cost for processing dimension δ_j is $C_j(n_j) \cdot \prod_{i \neq j} n_i$. 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 is equal to d times the size of the data cube if a one-dimensional pre-aggregation technique processes an array of size n_j at cost n_j.

3.3 Querying an Iterative Data Cube

Aggregate range queries as issued by a user or application select ranges on the *original* data cube. This data cube, however, was replaced by an Iterative Data Cube where cells contain pre-computed aggregate values. The query therefore needs to be translated to match the different contents. We will show that the problem of querying a high-dimensional IDC can be reduced to the one-dimensional cases.

Let Θ be a one-dimensional pre-aggregation technique, and let A and A_Θ denote the original and pre-aggregated data cubes, respectively. Technique Θ has to be *complete* in the sense that it must be possible to answer each range sum query on A by using A_Θ. Formally, for each range r on A there must exist coefficients $\beta_{r,l}$, such that

$$\sum_{j \in r} A[j] = \sum_{l=0}^{n-1} \beta_{r,l} A_\Theta[l] \tag{7}$$

where the $\beta_{r,l}$ are variables whose values depend on the pre-aggregation technique and the selected range. In the example in Fig. 1 the coefficients for SRPS (range $r = 2:5$) are $\beta_{r,0} = \beta_{r,1} = -1$, $\beta_{r,3} = \beta_{r,5} = 1$, and $\beta_{r,l} = 0$ for $l \in \{2, 4, 6, 7, 8\}$.

On a d-dimensional data cube A a range sum query selects a range r_i for each dimension δ_i. The answer Q to this query is computed as

$$Q = \sum_{j_d \in r_d} \sum_{j_{d-1} \in r_{d-1}} \cdots \sum_{j_1 \in r_1} A[j_1, j_2, \ldots, j_d] \ . \tag{8}$$

Recall, that the pre-aggregated cube A_d for A was obtained by iteratively applying one-dimensional pre-aggregation techniques, such that data cube A_i is computed by applying technique Θ_i along dimension δ_i to A_{i-1} (let $A_0 = A$). Consequently, range sum Q can alternatively be computed as

$$Q = \sum_{j_d \in r_d} \sum_{j_{d-1} \in r_{d-1}} \cdots \sum_{j_2 \in r_2} \Big(\sum_{l_1=0}^{n_1-1} \beta_{1,r_1,l_1} A_1[l_1, j_2, \ldots, j_d] \Big) \tag{9}$$

$$= \sum_{l_1=0}^{n_1-1} \beta_{1,r_1,l_1} \Big(\sum_{j_d \in r_d} \sum_{j_{d-1} \in r_{d-1}} \cdots \sum_{j_2 \in r_2} A_1[l_1, j_2, \ldots, j_d] \Big) \tag{10}$$

$$= \sum_{l_1=0}^{n_1-1} \beta_{1,r_1,l_1} \sum_{j_d \in r_d} \sum_{j_{d-1} \in r_{d-1}} \cdots \sum_{j_3 \in r_3} \Big(\sum_{l_2=0}^{n_2-1} \beta_{2,r_2,l_2} A_2[l_1, l_2, j_3, \ldots, j_d] \Big) \tag{11}$$

$$\vdots$$

$$= \sum_{l_1=0}^{n_1-1} \sum_{l_2=0}^{n_2-1} \cdots \sum_{l_d=0}^{n_d-1} \beta_{1,r_1,l_1} \beta_{2,r_2,l_2} \cdots \beta_{d,r_d,l_d} A_d[l_1, l_2, \ldots, l_d] \ . \tag{12}$$

The β_{i,r_i,l_i} are well defined by the aggregation technique Θ_i and the selected range r_i. There are no dependencies between the different dimensions in the sense that β_{i,r_i,l_i} does not depend on the techniques Θ_j and the ranges r_j, if $j \neq i$. This enables the efficient decomposition into one-dimensional sub-problems. Note, that cell $A_d[l_1, \ldots, l_d]$ of the pre-aggregated array A_d contributes to the query result Q if and only if the value of $\beta_{1,r_1,l_1}\beta_{2,r_2,l_2} \cdots \beta_{d,r_d,l_d}$ is not zero.

The *query algorithm* follows directly from the above discussion. For each dimension δ_i and range r_i, the set of all l_i such that β_{i,r_i,l_i} is non-zero is determined independently of the other dimensions. Then, for each possible combination of non-zero β_{1,r_1,l_1}, β_{2,r_2,l_2},$\ldots$,β_{d,r_d,l_d} the cell $A_d[l_1, l_2, \ldots, l_d]$ has to be accessed and contributes its value, multiplied by $\beta_{1,r_1,l_1}\beta_{2,r_2,l_2} \cdots \beta_{d,r_d,l_d}$, to the final result Q of the range sum query.

Figure 3 shows an example for a query that computes $\mathtt{SUM}(A[2,4] : A[5,6])$ on a two-dimensional pre-aggregated data cube (SRPS with box size 3 applied along both dimensions). First, for range $2:5$ in dimension δ_1 and range $4:6$ in dimension δ_2 the indices with non-zero β values are obtained together with the βs. Recall, that for range $r_1 = 2:5$ we obtained the values $\beta_{1,r_1,0} = \beta_{1,r_1,1} = -1$, $\beta_{1,r_1,3} = \beta_{1,r_1,5} = 1$, and $\beta_{1,r_1,l_1} = 0$ for $l_1 \in \{2,4,6,7,8\}$ (see above). Similarly, we obtain $\beta_{2,r_2,3} = -1$, $\beta_{2,r_2,6} = 1$, and $\beta_{2,r_2,l_2} = 0$ for $l_2 \in \{0,1,2,4,5,7,8\}$ for range $r_2 = 4:6$ in dimension δ_2. Combining the results leads to the correct computation of $\mathtt{SUM}(A[2,4] : A[5,6])$ as $A_2[0,3] - A_2[0,6] + A_2[1,3] - A_2[1,6] - A_2[3,3] + A_2[3,6] - A_2[5,3] + A_2[5,6]$.

The *query cost* of IDC, i.e., the number of cells accessed in A_d, follows directly from the algorithm. It is the product of the sizes of the sets of non-empty β values obtained for each dimension. As a consequence, once the worst case or average query cost of a one-dimensional technique is known, it is easy to compute the worst/average query cost for the d-dimensional pre-aggregated data cube by multiplying the one-dimensional costs. In our example, one-dimensional SRPS allows each range sum to be computed from at most 4 values (computing $\mathtt{SUM}(A[e] : A[f])$ with SRPS requires at most accessing the anchor of the box that contains f, cell f, and, if $e > 0$, the anchor of the box that contains $e-1$ and cell $e-1$). Consequently, independent of the selected ranges at most 4^d cells in the d-dimensional pre-aggregated SRPS data cube have to be accessed.

3.4 Updating an Iterative Data Cube

For the original data cube, an update to the data set only affects a single cell. Since Iterative Data Cubes store pre-computed aggregates, such an update has to be translated to updates on a set of cells in the pre-aggregated data cube. The set of affected cells in A_d follows directly from (6). Note that equations (6) and (12) are very similar, therefore the algorithms for processing queries and updates are almost identical.

Equation (1) describes the dependencies between the pre-aggregated and the original data cube for the one-dimensional case. Clearly $A_\Theta[j]$ is affected by an update to $A[k]$ if and only if $\alpha_{j,k} \neq 0$ (see

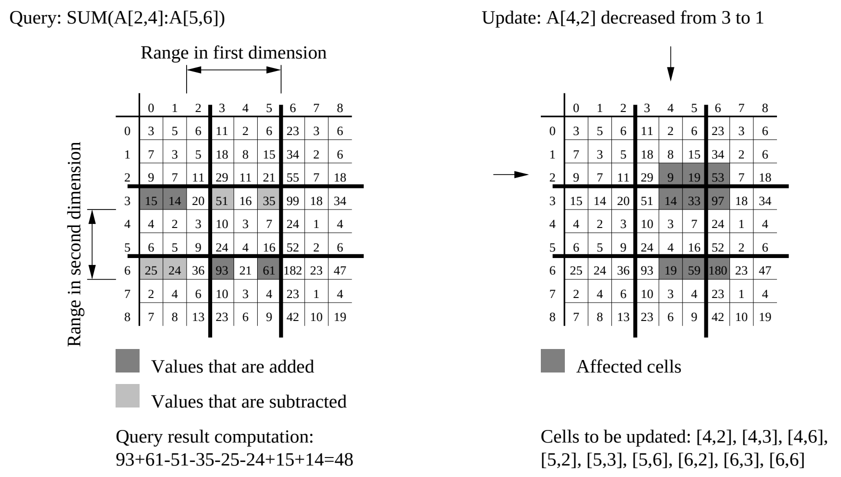

Query: SUM(A[2,4]:A[5,6])
Range in first dimension
Range in second dimension
Values that are added
Values that are subtracted
Query result computation:
93+61-51-35-25-24+15+14=48
Update: A[4,2] decreased from 3 to 1
Affected cells
Cells to be updated: [4,2], [4,3], [4,6], [5,2], [5,3], [5,6], [6,2], [6,3], [6,6]

Fig. 3. Processing queries and updates on an Iterative Data Cube (SRPS technique used for both dimensions)

Fig. 1 for an example). Based on (6) this can be generalized to d dimensions. Let $[k_1, \ldots, k_d]$ be the cell in the original data cube A which is updated by a value Δ. For each dimension δ_i, the set of all c_i such that α_{i,c_i,k_i} is non-zero is determined independently of the other dimensions. Then, for each possible combination of non-zero α_{1,c_1,k_1}, α_{2,c_2,k_2},$\ldots$,α_{d,c_d,k_d} the cell $A_d[c_1, c_2, \ldots, c_d]$ has to be updated by $\Delta \cdot \alpha_{1,c_1,k_1}\alpha_{2,c_2,k_2} \cdots \alpha_{d,c_d,k_d}$.

Figure 3 shows an example for an update that decreases the value $A[4,2]$ in the original data cube by 2. Recall, that SRPS with block size $s = 3$ was applied along both dimensions of the data cube and that the corresponding coefficients are $\alpha_{1,0,0} = 1$, $\alpha_{1,1,1} = 1$, $\alpha_{1,2,1} = \alpha_{1,2,2} = 1$, $\alpha_{1,3,k_1} = 1$ for $0 \leq k_1 \leq 3$, $\alpha_{1,4,4} = 1$, $\alpha_{1,5,4} = \alpha_{1,5,5} = 1$, $\alpha_{1,6,k_1} = 1$ for $0 \leq k_1 \leq 6$, and $\alpha_{1,c_1,k_1} = 0$ for all other combinations of c_1 and k_1. In dimension δ_1 the updated cell has the index value 4, i.e., the relevant coefficients are $\alpha_{1,4,4}$, $\alpha_{1,5,4}$, and $\alpha_{1,6,4}$ which have the value 1, while all other $\alpha_{1,c_1,4}$ are zero. Similarly the non-zero coefficients $\alpha_{2,2,2} = \alpha_{2,3,2} = \alpha_{2,6,2} = 1$ are obtained. Consequently, the cells $[4,2]$, $[4,3]$, $[4,6]$, $[5,2]$, $[5,3]$, $[5,6]$, $[6,2]$, $[6,3]$, and $[6,6]$ in A_2 have to be updated by $1 \cdot (-2)$.

The *update cost* of IDC, i.e., the number of accessed cells in A_d, is the product of the sizes of the sets of non-empty α values obtained for each dimension. Thus, like for the query cost, once the worst case or average update cost of a one-dimensional technique is known, it is easy to compute the worst/average update cost for high-dimensional Iterative Data Cubes. This is done by multiplying the worst/average update costs of the one-dimensional techniques.

4 IDC for Real-World Applications

We present one-dimensional aggregation techniques and discuss how they are selected for pre-aggregating a high-dimensional data cube. The presented techniques mainly illustrate the range of possible tradeoffs between query and update cost. In the following discussion the original array is denoted with A and has n elements $A[0]$, $A[1]$,$\ldots$,$A[n-1]$. The pre-aggregated array will be named like the corresponding generating technique.

4.1 One-Dimensional Pre-Aggregation Techniques

The pre-aggregated array used for the PS technique [11] contains the prefix sums of the original array, i.e., $\mathrm{PS}[j] = \sum_{k=0}^{j} A[k]$. Figure 1 shows an example for $n = 9$. Any range sum on A can be computed by accessing at most two values in PS (difference between value at endpoint and predecessor of anchor of query range). On the other hand, an update to $A[k]$ affects all $\mathrm{PS}[j]$ where $j \geq k$. This results in worst case costs of 2 for a query and of n for an update. In Fig. 1 cells in PS which have to be accessed in order to answer `SUM`$(A[2] : A[5])$ and those that are affected by an update to $A[4]$ are shaded.

The SRPS technique [16] (Fig. 1) was already introduced in Sect. 3.2. Its worst case costs are 4 for queries, and $2\sqrt{n}$ (or $2\sqrt{n}-2$ when n is a perfect square) for updates [16].

To compute the pre-aggregated array SDDC, the SDDC technique [16] first partitions the array A into two blocks of equal size. The anchor cell of each block stores the corresponding prefix sum of A. For each block, the same technique is applied recursively to the sub-arrays of non-anchor cells. The recursive partitioning defines a hierarchy, more precisely a tree of height less or equal to $\lceil \log_2 n \rceil$, on the partitions (blocks). Queries and updates conceptually descend this tree. The processing starts at the root and continues to that block that contains the endpoint of the query or the updated cell, respectively. A query $\mathtt{SUM}(A[0] : A[c])$ is answered by adding the values of the anchors of those blocks that contain c. Due to the construction, at most one block per level can contain c, resulting in a worst case prefix sum query cost of $\lceil \log_2 n \rceil$. Queries with ranges $[x] : [y]$ where $x > 0$ are answered as $\mathtt{SUM}([0] : [y]) - \mathtt{SUM}([0] : [x-1])$. Thus the cost of answering any range sum query is bounded by $2\lceil \log_2 n \rceil$. At each level an update to a cell u in a block U only propagates to those cells that have a greater or equal index than u and are an anchor of a block that has the same parent as U. Consequently, the update cost is bounded by the height of the tree ($\lceil \log_2 n \rceil$). In Fig. 4 an example of an SDDC array and how the query $\mathtt{SUM}(A[2] : A[5])$ and an update to $A[4]$ are processed are shown. Note, that SDDC can be generalized by choosing different numbers of blocks and different block sizes when partitioning the data cube. This enables the technique to take varying attribute hierarchies into account.

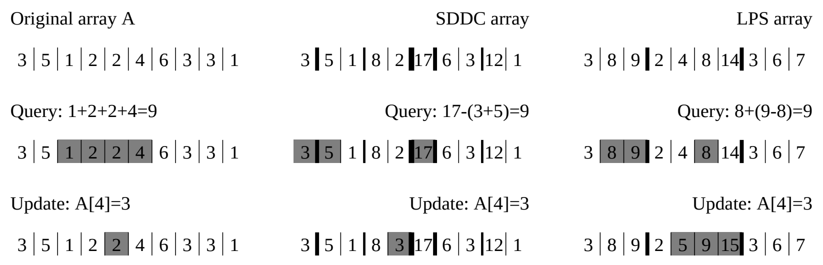


Fig. 4. Original array A and corresponding SDDC and LPS ($s_1 = 3$, $s_2 = 4$, $s_3 = 3$) arrays (query range and updated cell are framed, accessed cells are shaded)

The Local Prefix Sum (LPS) technique partitions array A into t blocks of sizes $s_1, s_2, \ldots, s_t$, respectively. Any cell in the pre-aggregated array LPS contains a "local" prefix sum, i.e., the sum of its value and the values in its left neighbors until the anchor of the block it is contained in. A range query is answered by adding the values of all block endpoints that are contained in the query range, adding to it the value of the cell at the endpoint of the query range (if it is not an endpoint of a block) and subtracting the value of the cell left to the anchor of the query range (if it is not an endpoint of a block). Thus the query cost is bounded by $t+1$. Figure 4 shows an example. Updates only affect cells with a greater or equal index than the updated cell in the same block, resulting in a worst case update cost of $\max\{s_1, \ldots, s_t\}$. For a certain t the query cost is fixed, but the worst case update cost is minimized by choosing $s_1 = s_2 = \ldots = s_t$. The corresponding family of (query cost, update cost) tradeoffs therefore becomes $(t+1, \lceil n/t \rceil)$.

The two techniques of using A directly or using its prefix sum array PS instead, constitute the extreme cases of minimal cost of updates and minimal cost of queries for one-dimensional data. Note, that it is possible to reduce the worst case query cost to 1. This, however, requires pre-computing and storing the result for any possible range query, i.e., $\frac{n}{2}(n+1)$ values. Also, since $A[\lfloor n/2 \rfloor]$ is contained in $(\lfloor n/2 \rfloor + 1)(n - \lfloor n/2 \rfloor)$ different ranges, the update cost for this scheme is at least $n^2/4$. Since we focus on techniques that do not introduce space overhead, PS is the approach with the minimal query cost. Table 1 summarizes the query and update costs for selected one-dimensional techniques.

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	$2\sqrt{n}-2$	when n perfect square
	4	$2\sqrt{n}$	otherwise
Space-Efficient Dynamic Data Cube (SDDC)	$2\lceil\log_2 n\rceil$	$\lceil\log_2 n\rceil$	
Local Prefix Sum (LPS)	$t+1$	$\lceil n/t\rceil$	$2 \le t < n$

Table 1. Query-update cost tradeoffs for selected one-dimensional techniques

4.2 Selecting an IDC for an Application

The IDC technique provides a modular framework for choosing a suitable pre-aggregation scheme. It greatly simplifies taking advantage of a priori knowledge about an application. For instance, when it is known that a hierarchy exists for an attribute and that users typically query according to this hierarchy (e.g., it is more likely that a query aggregates monthly sales figures than sales figures for a 30-day period that starts in the middle of a month), one can set a corresponding block size for SDDC or SRPS. If a dimension has only a few values (e.g., gender), the best choice in most cases will be PS or not pre-aggregating along this dimension at all. Alternatively, if no appropriate technique is available, it is relatively easy to develop a new one and to integrate it into the framework. Recall, that all one has to do is to develop a *one*-dimensional technique and to analyze its cost tradeoffs.

The process of selecting an appropriate IDC is illustrated with a hypothetical example. Assume that the data cube has three dimensions of size n each, and a fourth dimension of size 2 (e.g., gender). Two of the three attributes with dimension size n are hierarchical and it is likely that users query according to the hierarchies. For simplicity assume further that both hierarchies are similar to a balanced binary tree. Apart from that, the query cost has to be small, but frequent updates are expected. Then the best choice for the two hierarchical attributes is the SDDC technique (depending on the actual hierarchical structure, variations of SDDC can be used). It guarantees a sublinear query and update cost and provides good expected costs for queries that aggregate according to the hierarchies. For the dimension of size 2 pre-aggregation is unnecessary. The remaining dimension is processed with PS to enable fast queries. In total, the worst case costs are $2\log_2 n \cdot 2\log_2 n \cdot 2 \cdot 2 = 16\log_2^2 n$ for queries and $\log_2 n \cdot \log_2 n \cdot 1 \cdot n = n\log_2^2 n$ for updates. Note that all costs are *exact*, i.e., there are no hidden constants.

5 Comparing IDC to Previous Approaches

The IDC technique reduces the problem of pre-aggregating d-dimensional data cubes to the one-dimensional case. Compared to techniques that directly solve the d-dimensional problem, IDC's range of possible query-update cost tradeoffs is therefore restricted. However, as we will show below, none of the previously proposed d-dimensional pre-aggregation techniques obtains superior tradeoffs.

The PS, SRPS, and SDDC techniques constitute special cases of IDC. One can iteratively create the pre-aggregated data cubes for these techniques by applying the corresponding one-dimensional technique for each dimension of the original data cube. This results in d-dimensional Iterative Data Cubes with worst case (query, update)-cost pairs $(2^d, n^d)$, $(4^d, 2^d n^{d/2})$, and $(2^d\lceil\log_2 n\rceil^d, \lceil\log_2 n\rceil^d)$, respectively. As an interesting by-product PS, SRPS, and SDDC can be analyzed and implemented as Iterative Data Cubes. Note that for SRPS and SDDC the implementation is quite complex and the analysis difficult. For instance for a d-dimensional data cube, SDDC stores $(d-1)$-dimensional surfaces of pre-aggregated cumulative values recursively as $(d-1)$-dimensional data cubes. Thus, IDC provides a great "tool" for verifying the results of these previous approaches and for obtaining new results, like for instance average case costs.

The HC technique [3] generates a pre-aggregated data cube by hierarchically partitioning the original data cube A into smaller hyper-rectangles (blocks) of equal size. The number of recursive partitioning steps determines the height of a Hierarchical Cube. Hierarchical Rectangle Cubes (HRC) with a height of one are identical to the original data cube. In HRCs of height two each cell stores the prefix sum local to the anchor of the block it belongs to. Consequently, any HRC of height two can be constructed iteratively by applying the one-dimensional LPS technique with the corresponding block sizes along each dimension. Hierarchical Rectangle Cubes of height one and two hence are generalized by IDC. For HRCs of height greater than two, [3] does not provide analytical or experimental results. Thus we were not able to compare IDC to HRCs of height greater than two. Hierarchical Band Cubes (HBC) can not be generalized by our technique. Only HBCs of height one are identical to the PS cube, which is an IDC. For HBCs of height greater than one we prove, that no matter which hierarchical partitioning scheme is used, a d-dimensional HBC has always a worst case update cost of at least n^{d-1} (we assume without loss of generality that all dimensions have a domain of size n). The proof can be found in the appendix. The range of possible update costs therefore is restricted compared to IDC. In total, the best possible HBC cube of height $h \geq 2$ has a worst case query cost of at least $2^d h = 2^{d+1}$ [3], and a worst case update cost of at least n^{d-1}. An Iterative Data Cube where the PS technique is used for $(d-2)$ dimensions and the SRPS technique is used for the remaining two dimensions, has respective worst case query and update costs of $2^{d-2}4^2 = 2^{d+2}$ and $n^{d-2}(2\sqrt{n})^2 = 4n^{d-1}$. Thus, there exists an IDC whose query and update costs are asymptotically identical to the *lower bounds* for the corresponding costs of any HBC cube.

6 Conclusion

IDC is the first pre-aggregation technique on data cubes that can take the specific properties of different dimension attributes into account. Instead of solving a d-dimensional pre-aggregation problem directly, the different dimensions are handled independently. This greatly simplifies the development, analysis, and implementation compared to earlier approaches. At the same time a greater variety of query-update cost tradeoffs can be generated. Thus Iterative Data Cubes provide a practical framework for developing pre-aggregation techniques for MOLAP data cubes.

Even though the space of possible pre-aggregation schemes is restricted by the iterative combination process, we were able to show that the query-update cost tradeoffs of previously proposed techniques are matched. It remains, however, as an open problem, to show that in general the query-update cost tradeoffs that are optimal for the IDC technique are also optimal with respect to any pre-aggregation technique on a high-dimensional data cube. We will pursue this problem, as well as the problem of sparse data sets [15] in our future research.

References

[1] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 156–165, 1997.

[2] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg CUBEs. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 359–370, 1999.

[3] C.-Y. Chan and Y. E. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 675–686, 1999. Extended version published as Tech. Report, Univ. of Wisconsin, 1999.

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

[5] S. Geffner, D. Agrawal, and A. El Abbadi. The dynamic data cube. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, pages 237–253, 2000.

[6] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 328–335, 1999.

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

[8] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 98–112, 1997.

[9] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 208–219, 1997.

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 205–216, 1996.

[11] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 73–88, 1997.

[12] T. Johnson and D. Shasha. Some approaches to index design for cube forests. *IEEE Data Engineering Bulletin*, 20(1):27–35, 1997.

[13] Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 249–258, 1998.

[14] N. Pendse and R. Creeth. The OLAP report. http://www.olapreport.com/Analyses.htm, 2000. Parts available online in the current edition.

[15] M. Riedewald, D. Agrawal, and A. El Abbadi. pCube: Update-efficient online aggregation with progressive feedback and error bounds. In *Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 95–108, 2000.

[16] M. Riedewald, D. Agrawal, A. El Abbadi, and R. Pajarola. Space-efficient data cubes for dynamic environments. In *Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 24–33, 2000.

[17] J. R. Smith, V. Castelli, A. Jhingran, and C.-S. Li. Dynamic assembly of views in data cubes. In *Proc. Symp. on Principles of Database Systems (PODS)*, pages 274–283, 1998.

[18] Transaction Processing Performance Council. TPC-H benchmark (1.1.0). Available at http://www.tpc.org.

[19] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 193–204, 1999.

[20] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 96–104, 1998.

A Algorithm for Constructing an IDC

```
Algorithm Construction
Input: original data cube $A$, one-dimensional techniques $\Theta_1$, $\Theta_2$,..., $\Theta_d$
Output: pre-aggregated data cube $A_d$

$A_0 = A$;
For each dimension $\delta_i$ do
    For each $c_1 \in \{0, 1, \ldots, n_1 - 1\}$,...,$c_{i-1} \in \{0, 1, \ldots, n_{i-1} - 1\}$,
    $c_{i+1} \in \{0, 1, \ldots, n_{i+1} - 1\}$,..., $c_d \in \{0, 1, \ldots, n_d - 1\}$ do
        $(A_i[c_1, \ldots, c_{i-1}, 0, c_{i+1}, \ldots, c_d], A_i[c_1, \ldots, c_{i-1}, 1, c_{i+1}, \ldots, c_d], \ldots,$
        $A_i[c_1, \ldots, c_{i-1}, n_i - 1, c_{i+1}, \ldots, c_d]) =$
        $\Theta_i$.Construct($A_{i-1}[c_1, \ldots, c_{i-1}, 0, c_{i+1}, \ldots, c_d]$,
        $A_{i-1}[c_1, \ldots, c_{i-1}, 1, c_{i+1}, \ldots, c_d], \ldots,$
        $A_{i-1}[c_1, \ldots, c_{i-1}, n_i - 1, c_{i+1}, \ldots, c_d]$);

Function $\Theta_i$.Construct /* Defined for each one-dimensional technique $\Theta_i$ */
Input: array $B$ with $n$ values $B[0]$, $B[1]$,..., $B[n-1]$
Output: corresponding pre-aggregated array $B_i$

For $j = 0$ to $n - 1$ do
    $B_i[j] = \sum_{k=0}^{n-1} \alpha_{i,j,k} B[k]$;
Return ($B_i[0]$, $B_i[1]$,...,$B_i[n-1]$);
```

B Algorithm for Computing a Range Sum Query on IDC

```
Algorithm Query
Input: pre-aggregated data cube $A_d$; ranges $r_1$, $r_2$,..., $r_d$
    (selected by query for original data cube $A$)
Output: range sum for selected range on $A$

Result = 0;
For each dimension $\delta_i$ do
    $S_i = \Theta_i$.QueryRange($r_i$); /* $S_i = \{(\beta_{i,r_i,l_i}, l_i) | \beta_{i,r_i,l_i} \neq 0\}$ */
For each $(\beta_{1,r_1,l_1}, l_1) \in S_1$ do
    For each $(\beta_{2,r_2,l_2}, l_2) \in S_2$ do
        ⋮
        For each $(\beta_{d,r_d,l_d}, l_d) \in S_d$ do
            Result = Result + $\beta_{1,r_1,l_1} \beta_{2,r_2,l_2} \cdots \beta_{d,r_d,l_d} A_d[l_1, l_2, \ldots, l_d]$;
Return (Result);

Function $\Theta_i$.QueryRange /* Defined for each one-dimensional technique $\Theta_i$ */
Input: range $r$
Output: translated "range" with scaling factors, i.e., $\{(\beta_{i,r,l}, l) | \beta_{i,r,l} \neq 0\}$

$S = \{\}$;
For all $l$ do
    If ($\beta_{i,r,l} \neq 0$) then $S = S \cup \{(\beta_{i,r,l}, l)\}$;
Return (S);
```

C Algorithm for Updating an IDC

```
Algorithm Query
Input: pre-aggregated data cube $A_d$; updated cell $u = [u_1, \ldots, u_d]$ in original
    data cube; difference $\Delta$ between new and old value of $u$
Output: updated data cube $A_d$

For each dimension $\delta_i$ do
    $S_i = \Theta_i$.UpdateRange($u_i$); /* $S_i = \{(\alpha_{i,c_i,u_i}, c_i) | \alpha_{i,c_i,u_i} \neq 0\}$ */
For each $(\alpha_{1,c_1,u_1}, c_1) \in S_1$ do
    For each $(\alpha_{2,c_2,u_2}, c_2) \in S_2$ do
        ⋮
        For each $(\alpha_{d,c_d,u_d}, c_d) \in S_d$ do
            $A_d[c_1, c_2, \ldots, c_d] = A_d[c_1, c_2, \ldots, c_d] + \alpha_{1,c_1,u_1}\alpha_{2,c_2,u_2} \cdots \alpha_{d,c_d,u_d} \cdot \Delta$;
Return ($A_d$);

Function $\Theta_i$.UpdateRange /* Defined for each one-dimensional technique $\Theta_i$ */
Input: index $u_i$ of updated cell
Output: "range" of affected cells' indices with scaling factors, i.e.,
    $\{(\alpha_{i,j,u_i}, j) | \alpha_{i,j,u_i} \neq 0\}$
$S = \{\}$;
For all $j$ do
    If $(\alpha_{i,j,u_i} \neq 0)$ then $S = S \cup \{(\alpha_{i,j,u_i}, j)\}$;
Return (S);
```

D A Lower Bound for the Update Cost on Hierarchical Band Cubes

We prove a lower bound on the worst case update cost for any HBC of height two or greater. Recall, that A denotes a d-dimensional original data cube. Without loss of generality let each dimension have a domain $\{0, 1, \ldots, n-1\}$. Also, let A_{HBC} be a corresponding HBC data cube of height $h \geq 2$. We will show, that the worst case update cost on A_{HBC} is at least n^{d-1}. The proof bases on the fact that there are regions of size n^{d-1} in the HBC cube where the value of each cell in the region depends on a certain cell in the original data cube.

D.1 Notation for the Proof

For the proof, we first need to introduce additional notation which is similar to the notation used in [3]. Let δ be a dimension attribute with domain $\{0, \ldots, n-1\}$ and $\langle b_h, b_{h-1}, \ldots, b_1 \rangle$ be a *base sequence* of positive integers, such that $n = \prod_{i=1}^{h} b_i$. An integer $v \in$ domain(δ) can be decomposed into a sequence of h component values $\langle v_h, v_{h-1}, \ldots, v_1 \rangle$ such that

$$v = v_h \sum_{j=1}^{h-1} b_j + \ldots + v_i \sum_{j=1}^{i-1} b_j + \ldots + v_2 b_1 + v_1 \quad . \tag{13}$$

The v_i are base-b_i digits. For example, if $n = 8$, then 5 is decomposed into $\langle v_1, v_2, v_3 \rangle = \langle 1, 0, 1 \rangle$ according to base sequence $\langle b_1, b_2, b_3 \rangle = \langle 2, 2, 2 \rangle$. Similarly, 6 is decomposed into $\langle 1, 1, 0 \rangle$. On a d-dimensional data cube, base sequence $\langle b_{i,h}, b_{i,h-1}, \ldots, b_{i,1} \rangle$ is used to decompose dimension δ_i. Intuitively, the data cube is recursively partitioned into smaller blocks. For instance, let $A[0,0] : A[7,7]$ be a two-dimensional data cube. If base sequence $\langle 2, 2, 2 \rangle$ is used for both dimensions, then the (8×8)-data cube is first partitioned into 4 blocks of size 4×4 each. These (4×4)-blocks are in turn partitioned into 4 (2×2)-sub-blocks each, which are similarly split into 4 single cells. The *rank* of a block is defined as follows. The blocks that are the result of the first partitioning are of rank h. Their children have rank $h-1$, and so on. Formally, the blocks of rank h have ranges of size $\prod_{k=1}^{h-1} b_{i,k}$ in dimension δ_i. A rank-l block is partitioned into $\prod_{i=1}^{d} b_{i,l-1}$ rank-$(l-1)$ sub-blocks of range-size $\prod_{k=1}^{l-2} b_{i,k}$ in dimension δ_i.

The anchors of the rank-h blocks are referred to as *root* cells ([0,0], [0,4], [4,0], [4,4] in the example). The *level* of a cell c is the greatest number k such that c is the anchor of a rank-k block. In the example, the root cell [0,0] is at level 3, cell [4,2] is at level 2, [3,0] is at level 1. The level of a cell $c = [c_1, \ldots, c_d]$ is formally defined as follows. Let $\langle c_{i,h}, c_{i,h-1}, \ldots, c_{i,1} \rangle$ be the decomposition of c_i according to $\langle b_{i,h}, b_{i,h-1}, \ldots, b_{i,1} \rangle$. Then level($c$) is defined by

1. $\text{level}(c) = h$ if $\forall 1 \leq i \leq d, 1 \leq j < h : \quad c_{i,j} = 0$
2. Otherwise, $\text{level}(c) = k$, if $\forall 1 \leq i \leq d, 1 \leq j < k : c_{i,j} = 0 \quad$ and $\quad \exists 1 \leq r \leq d : c_{r,k} > 0$.

Let $c = [c_1, \dots, c_d]$ be a cell at level $k < h$. Its *parent* $p = [p_1, \dots, p_d]$ is defined by $\forall 1 \leq i \leq d : \quad p_i = c_i - (c_i \bmod \prod_{j=1}^{k} b_{i,k})$. Intuitively, the parent of a cell c at level l is the anchor of that rank-$(l+1)$ block that contains c. In the example, $[2, 2]$ is the parent of $[2, 3]$, $[3, 2]$, and $[3, 3]$.

A Hierarchical Band Cube is constructed, such that root cells store the corresponding prefix sum of the original array ($A_{\text{HBC}}[\text{root}] = \texttt{SUM}(A[0, \dots, 0] : A[\text{root}])$). All other cells c aggregate the values of the cells in an "angular band" defined by $A_{\text{HBC}}[c] = \texttt{SUM}(A[0, \dots, 0] : A[c]) - \texttt{SUM}(A[0, \dots, 0] : A[\text{parent}(c)])$. With the defined notation we can now state the proof.

D.2 The Proof of the Lower Bound

First, we can assume that there exists a dimension δ_i, such that the last decomposition base $b_{i,1}$ is greater than 1. If such a dimension does not exist, all $b_{i,1}$ have the value 1 and therefore the blocks at rank 1 and rank 2 are identical. In such a case the redundant rank is removed, until the required property holds. Since we proof the property for HBCs of height two or greater, such a rank must exist (otherwise all bases in all dimensions are equal to 1, i.e., the HBC has height one). Consequently, the assumption holds and without loss of generality, we obtain $b_{1,1} > 1$, i.e., $b_{1,1} \geq 2$.

Now we examine all cells $c = [c_1, \dots, c_d]$ where c_2, c_3,..., c_d are allowed to be any values from the respective domains. The value c_1 is selected such that c_1 is not divisible by $b_{1,1}$. Note, that such a c_1 must exist because $b_{1,1} \geq 2$. The decomposition of c_1 according to the base sequence $\langle b_{1,h}, \dots, b_{1,1} \rangle$ is denoted with $\langle c_{1,h}, \dots, c_{1,1} \rangle$. Since c_1 is not divisible by $b_{1,1}$, the value $c_{1,1}$ in this decomposition is greater than zero (follows from (13)). Consequently, $\text{level}(c) = 1$. For the parent $p = [p_1, \dots, p_d]$ of cell c follows, that $p_1 = c_1 - (c_1 \bmod \prod_{j=1}^{1} b_{1,j}) = c_1 - (c_1 \bmod b_{1,1})$. Since c_1 is not divisible by $b_{1,1}$, we obtain that $p_1 < c_1$.

Consider an update that affects cell $u = [c_1, 0, 0, \dots, 0]$ in the original data cube. We show, that this update affects any cell c in A_{HBC} as selected above. First, since cell c is a level-1 cell, it can not be a root cell (root cells are level-h cells and $h \geq 2$). It therefore contains the value $\texttt{SUM}(A[0, \dots, 0] : A[c]) - \texttt{SUM}(A[0, \dots, 0] : A[p])$ (recall, that p is the parent of c). Obviously, the updated cell u is contained in region $A[0, \dots, 0] : A[c]$. On the other hand, since $p_1 < c_1$, u can not be contained in $A[0, \dots, 0] : A[\text{parent}(c)]$. Consequently, the value of u contributes to cell c in the HBC cube. Since the index values c_2, c_3,...,c_d can be selected arbitrarily, at least n^{d-1} cells c exist in the HBC cube that are affected by an update to u. This concludes the proof of the lower bound of n^{d-1} for the update cost on any HBC with two or more levels.