In Proceedings Int. Conf. on Data Warehousing and Knowledge Discovery DaWaK, pages 24–33, 2000

Space-Efficient Data Cubes for Dynamic Environments

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

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

1 Introduction

Data cubes 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.

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 Fig. 2), 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.

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 is not 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 Sect. 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 The SRPS and SDDC Techniques

In this section SRPS and SDDC are presented and compared to 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 $A[c]$. Without loss of generality let the domain of each dimension attribute i be $\{0, 1, \ldots, n-1\}$. $e : f$ 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 $[n-1, \ldots, n-1]$. 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 $[0, \ldots, 0]$. We will refer to a region that is anchored at $[0, \ldots, 0]$ as a *prefix region*; a query that selects such a region is a *prefix query*. Note that according to [7] any range sum can be computed by combining the range sums of up to 2^d (which is a constant) prefix regions. Thus the problem of computing the sum for an arbitrary range is reduced to the problem of efficiently computing prefix sum queries. We will therefore only describe how SRPS and SDDC solve this problem.

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. 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 reduces the problem of summarizing any possible range to the problem of summarizing and combining prefix regions. 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 cost of 2^d for prefix queries, a worst case cost of 2^{2d} for general range sum queries, 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 some settings.

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.

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.

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.

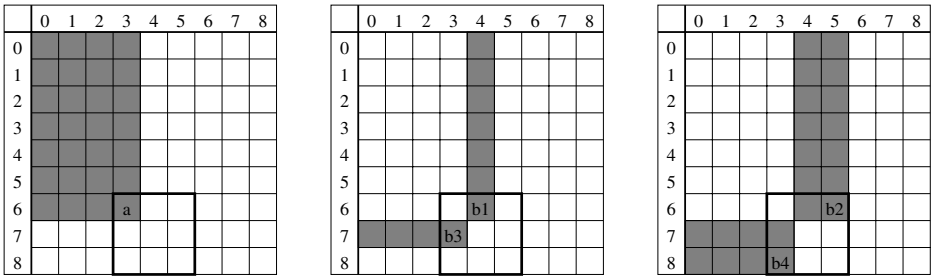


Fig. 1. Computation of border values as the sum of the values of the cells in the shaded area on array A

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

Original data cube

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

Cumulative cube for PS

	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

SRPS cube (3x3 boxes)

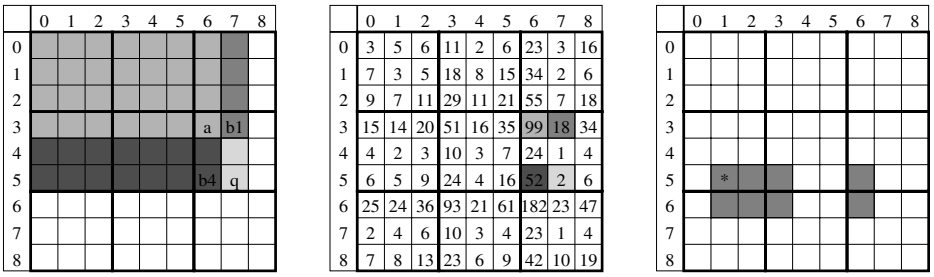
Fig. 2. Prefix Sum compared to SRPS

Any cell c in box B stores the value $\mathrm{SUM}(A[l_1, l_2, \ldots, l_d] : A[c])$ 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 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. 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 $\mathrm{SUM}(A[a_1 + 1, a_2 + 1, \ldots, a_d + 1] : A[c])$, which is the prefix sum relative to cell $[a_1 + 1, a_2 + 1, \ldots, a_d + 1]$. Figure 1 shows aggregation regions (shaded) for border cells of a two-dimensional data cube. In the example in Fig. 2 the original data cube and the corresponding SRPS cube are shown.

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



(a) Partitioning the query region (b) Query: 99+18+52+2=171 (c) Update to [5,1]

Fig. 3. Querying and updating an SRPS cube

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 $n^d(1 - (\frac{k-1}{k})^d)$.

Querying SRPS. As mentioned earlier, any range sum query can be answered by combining the results of up to 2^d appropriate prefix queries. Let $q = [q_1, \ldots, q_d]$ be the endpoint of the prefix region and $a = [a_1, \ldots, a_d]$ be the anchor of box B that contains q. Then the query region $[0, \ldots, 0] : q$ can be partitioned into non-overlapping regions which are identical to the aggregation regions of the cells in the set $\{c = [c_1, \ldots, c_d] | \forall i : 1 \leq i \leq d \wedge c_i \in \{a_i; q_i\}\}$. This set contains at most 2^d cells. Intuitively they are obtained as the "projection" of cell q to the surfaces of box B that contain the border cells (including cell q itself). Details about the partitioning are provided in [8]. Note, that the result for a prefix query can be obtained by adding values from a single box, which results in a high locality of accesses. Since a prefix query can be answered at a cost of 2^d, the *overall worst case range query cost* for SRPS becomes $2^d * 2^d = 2^{2d}$. Hence the query cost is constant irrespective of n, the size of the dimension domains of the data cube. Figure 3(a) shows an example for the partitioning of the query region for a two-dimensional data cube. The shaded cells in Fig. 3(b) need to be accessed in order to compute the prefix range sum.

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 3(c) shows an example. An update to cell $(5, 1)$ (marked with *) has to be propagated to each of the shaded cells.

To keep the description simple, we assume that k, the side-length of each box, evenly divides n, the side-length of the data cube. Clearly the number of cells that are affected by an update to a cell u is equal to the number of aggregation regions that contain u. From the definition of the aggregation regions follows that at most $(n/k + k - 2)^d$ aggregation regions contain cell u. This bound is tight; it is met when cell $u = [1, 1, \ldots, 1]$ is updated. Note, that the aggregation

regions that contain the updated cell are well defined. The cells that need to be updated are the endpoints of those regions. Details of the analysis are not provided here due to space limitations and can be found in [8].

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.[1] 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. Like for PS, RPS, and SRPS the answer to an arbitrary range sum query is obtained by combining the results of the corresponding prefix queries.

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. Each of the children are in turn subdivided into children, and so on.

The values in the border cells are cumulative. Thus an update to the anchor cell of a box has to be propagated to all border cells in the box. To reduce the update cost [3] introduces the B^c tree. 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, 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 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 cells are recursively stored as $(d-1)$-dimensional data cubes. Thus DDC with B^c trees and the recursive technique for storing the border values guarantees the polylogarithmic update and query costs of $\mathrm{O}(\log^d n)$.

2.4 SDDC: The Space-Efficient Dynamic Data Cube Technique

Like the Dynamic Data Cube, SDDC balances query and update costs to $\mathrm{O}(\log^d n)$. Those bounds are maintained 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.

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

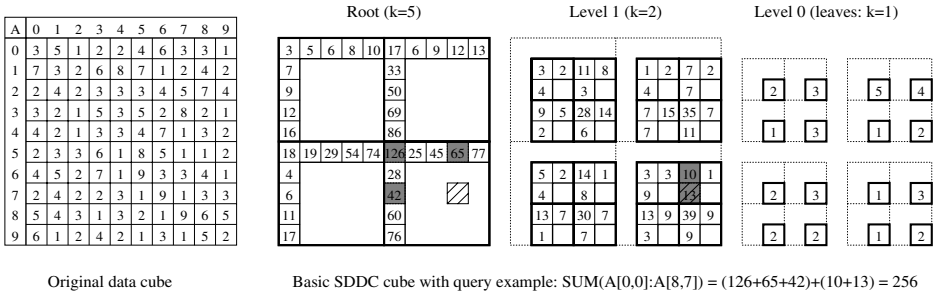


Fig. 4. Original data cube and corresponding SDDC cube

The Basic SDDC. To construct the SDDC cube, the data cube is first partitioned 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. 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. Their regions of inner cells are then in turn partitioned into 2^d boxes, and so on. Conceptually the boxes of the basic SDDC form a tree where each node corresponds to a box. The root node encompasses the entire data cube. The children of a node are those smaller boxes that partition the regions 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 4 presents a data cube and the corresponding basic SDDC.

Instead of dividing the region of the inner cells in each dimension in half, one could alternatively choose other partitionings into boxes. Partitions 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. In [8] we show that the basic DDC requires 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!

To find the sum for any prefix region, the tree is descended and the appropriate border values are added. On a tree level the query is answered as described for SRPS. The only difference is that instead of accessing an inner cell, the query recursively accesses the corresponding child node (see [8] for details). In the example in Fig. 4 the cells that are accessed in order to compute SUM($A[0,0] : A[8,7]$) are shaded; cell $(8,7)$ is hatched. Since the partitioning of

the data cube is non-overlapping, at most one box per level contains the endpoint of the query region. For each such box at most $2^d - 1$ border cells have to be accessed (analysis identical to SRPS). Since the tree has at most $\log_2 n$ levels, the cost for any prefix query 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 $[1, 1, \ldots, 1]$. This update already affects $\mathrm{O}(n^{d-1})$ border cells in the root node (see [8] for details).

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 $\mathrm{B^c}$ trees for balanced update and query costs on two-dimensional data cubes, and storing the border values of higher-dimensional data cubes recursively (see Sect. 2.3). However, $\mathrm{B^c}$ trees and the recursive approach introduce unnecessary redundancy. We follow a similar approach, but remove the additional storage requirements.

Recall that the values of the border cells in the same surface are cumulative, which results in the high worst case update costs. To reduce the costs for one-dimensional arrays of border cells we use an elegant technique that embeds a tree into the array. The main idea is to first replace the cumulative values by the corresponding differences of the values of neighboring cells and then to apply the basic SDDC technique to this array of differences. Queries and updates are processed as described for the basic SDDC technique, resulting in a worst case cost of $\mathrm{O}(\log n)$ for both operations. Thus the $\mathrm{B^c}$ tree is replaced by a data structure which does not add any space overhead compared to storing the original array.

The DDC technique stores d-dimensional surfaces of border cells recursively as $(d-1)$-dimensional data cubes. Since the surfaces are overlapping, redundancy is introduced. SDDC removes this redundancy by ensuring that values in the overlapping regions are stored only once. The idea is to embed the recursively computed values into the space of exactly those values they replace.

Note that improving on the $\mathrm{B^c}$ trees and the recursive technique for storing the values of the border cells further increases the space savings of SDDC compared to DDC. Due to the improvements, SDDC has the *same storage consumption as the original data cube*. Its query and update costs are $\mathrm{O}(\log^d n)$, which is sublinear in the side-length of the data cube. A more detailed description of the technique can be found in [8].

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 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.

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 I/O 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. Also, we intend to perform experiments to compare our techniques to previous approaches in more detail.

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.

[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.

[8] 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.