

# Space-Efficient Range-Sum Queries in OLAP

Fredrik Bengtsson and Jingsen Chen

Department of Computer Science and Electrical Engineering  
Luleå University of Technology S-971 87 LULEÅ, Sweden

**Abstract.** In this paper, we present a fast algorithm to answer range-sum queries in OLAP data cubes. Our algorithm supports constant-time queries while maintaining sub-linear time update and using minimum space. Furthermore, we study the trade-off between query time and update time. The complexity for query is  $O(2^{\ell d})$  and for updates  $O((2^{\ell} \sqrt[\ell]{n})^d)$  on a data cube of  $n^d$  elements, where  $\ell$  is a trade-off parameter. Our algorithm improve over previous best known results.

## 1 Introduction

In a large multidimensional database, aggregate queries can potentially consume enormous amount of time to process, if not properly supported by the database. The responsiveness of both updates to the database and queries from the database can be very poor. This is often very unsatisfactory, for example in an interactive environment, where the user needs a fast response from the database. To support the interactive analysis of large data sets, the On-Line Analytic Processing (OLAP) model [1, 2] has been used and received much attention.

In OLAP, the data consists of records with  $d + 1$  attributes. One attribute is the measure attribute and the others are called dimensional attributes. Thus, the database can be regarded as a  $d$ -dimensional cube. The data cube [3] is a popular data model for OLAP that allows the implementation of several important queries. Several query types have previously been studied, including range max/min [4, 5] and range-sum [6–10, 1].

In this paper, we study the orthogonal range-sum query on the data cube. Ho et al. [11] introduced the prefix sum technique for fast range-sum queries. The update time, however, is  $O(n^d)$  for data cubes of  $n^d$  elements. Nevertheless, the prefix sum techniques provides a base for further improvements [9, 6–8, 10]. Poon [12] has a very efficient structure with a query time of  $O((2L)^d)$  and update time of  $O((2Ln^{1/L})^d)$ , where  $L$  is a parameter. The structure uses  $O((2n)^d)$  storage. Geffner et al. [8] have a structure with with a query complexity of  $O(2^d)$  and an update complexity of  $O((n^{d/2})^d)$ . The space usage in this structure is also super linear. Chun et al. [6] presents a structure which uses a standard prefix sum and accumulates updates in an R-tree. Their structure has, according to their simulations, good performance, but the space usage is super linear. All the above-mentioned structures uses extra space in order to achieve high performance for both lookups and queries.

However, a structure saving space is the *Space Efficient Relative Prefix Sum* by Riedewald et al. [9]. It does not use any extra space, but has the same performance as the *Relative Prefix Sum*. Another structure that does not use extra space is the *Space Efficient Dynamic Data Cube* by Riedewald et al. [9]. It has logarithmic search complexity and logarithmic update complexity.

An interesting special case is when the data cube is considered to be sparse, and thus, the space savings could be huge if the sparsity is taken into account. This have been studied by Chun et al. [7]. They propose a technique called *pc-pool* for sparse cubes. They demonstrate the performance of the pc-pool by simulations.

Our results does not use any extra space, has constant query complexity while maintaining a sub-linear time update. Furthermore, our results provides superior query-update trade-off over previous known structures.

The rest of the paper is organized as follows. Section 2 introduces the prefix sum cube and methods which are closely related to our structure. We present our results together with their performance analysis in Section 3. The next section will investigate the Space Efficient Dynamic Data Cube structure further. First, we will analyse the complexity of the Space Efficient Dynamic Data Cube which are omitted in the original paper.[9]. Then, a study of the query-update complexity trade-off for the Space Efficient Dynamic Data Cube is also pursued. Section 5 presents some further improvements to our structure and Section 6 concludes the paper.

## 2 Prefix Sum Methods

This section will introduce methods that are closely related to our structure. All those structures are based on the prefix sum [11]. Our work is a generalization of both the Space-Efficient Relative Prefix Sum [9] and the Space-Efficient Dynamic Data Cube [9]. The Space-Efficient Relative Prefix Sum can be derived as a special case of our method.

Consider the two extremes: the original array and the prefix sum array. A query in the orthogonal array takes  $O(n)$  time, in the worst case. An update, on the other hand, requires only  $O(1)$  time. For an array of prefix sums, it's the opposite. Different techniques combines the best of those two approaches.

In the Relative Prefix Sum (RPS) [8], the data set is represented by two arrays; the *Overlay* array ( $l$ ) and the *Relative prefix* array ( $r$ ). Each array is divided in  $\sqrt{n_i}$  blocks of  $\sqrt{n_i}$  elements each, along dimension  $i$ , for  $i = 0, 1, \dots, d - 1$ .

In Fig. 1, a simple one dimensional example of the RPS is shown, where  $n = 16$  and the block boundaries have been emphasized with thick lines.

In  $r$ , the prefix sum relative to each block is stored.

In  $l$ , the first element of each block stores the sum of all elements (from the data cube) of all previous blocks. In  $d$  dimensions, the situation is slightly more complicated, but the idea is the same. Thus, there is no need to rebuild the whole array on update. It is sufficient to rebuild the block of  $r$  and all the elements in  $l$ , both of which have a size of  $O(\sqrt{n})$ .

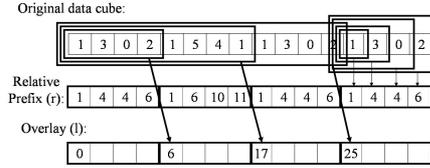


Fig. 1. Simple 1-dim RPS

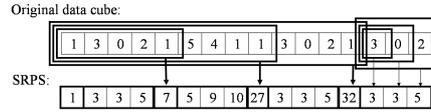


Fig. 2. Simple 1-dim SRPS

The Space-Efficient Relative Prefix Sum (SRPS) by Riedewald et al. [9] is a structure very similar to the RPS, but the space usage is smaller. Consider the one-dimensional RPS of Fig. 1 again. If the elements in the overlay array contains the inclusive sum (instead of the exclusive), we need only store a shorter local prefix sum in the RPS array. It is sufficient that the local prefix sum in the RPS array starts at the 2:nd position in the block (if the overlay sums are inclusive). The first position in the local prefix sum in the RPS array can then be used to store the overlay array. This way, the structure does not use more space than the original data cube (Fig. 2). Search and update is performed similar to the RPS.

In the Space-Efficient Dynamic Data Cube [9], the technique of the SRPS is applied recursively on each block and on the boundary cells of each block. However, instead of splitting the array in  $\sqrt{n_i}$  blocks, each dimension is split in 2 (or any other constant). In general, we get  $2^d$  boxes for a  $d$ -dimensional cube. Now, call the region corresponding to the prefix sum of a block, the *inner region* of each block. The inner region, in  $d$  dimensions, is all elements in a box, except for the first element in *each dimension*.

The inner region is a prefix sum, but consider the data cube that would have generated the prefix sum (the pairwise difference of the prefix sum). It is possible to recursively apply the SDDC algorithm to that data. Instead of storing the prefix sum in the inner region, we store the data from the recursive application of the SDDC. The recursion continues  $O(\log n)$  levels.

Now, consider the border cells of a box (cells that are not the inner region – the first cell of each block, in the one-dimensional case). Observe that, in the one-dimensional case, the border of each block has dimension 0 (it is a scalar). In general, for  $d$  dimensions, the border cells contains  $\binom{d}{k}$  regions of  $k$  dimensions. Each element (border cell) is the sum of all elements in the data cube with smaller index. Each such sum span  $d - k$  dimensions. For each region of the border of each box, the SDDC algorithm (of the dimensionality of the region) is applied recursively. The elements are stored in the same place as we would have done without recursion, thus, throwing away the old elements (from before the recursion). Queries and updates are handled similar to the SRPS, but recursively.

This way, we get a structure (and a corresponding algorithm) requiring  $O(\log^d n)$  array accesses for both update and query. See Section 4 for a performance analysis, which is omitted in the original paper [9].

### 3 Our result

Our structure for range sum queries combine the techniques from both the SRPS and the SDDC. We use a recursive storage technique to obtain fast range sum queries (constant time) and to achieve good trade-off between queries and updates. In essence, instead of storing the elements from smaller subproblems that appear in the structure directly, our algorithm is applied recursively and the result from the recursion is stored. This results in a query performance of  $O(2^{id})$  while maintaining an update complexity of  $O((2^i \sqrt[n]{n})^d)$ , where  $i$  is a trade-off parameter. This improves over previous known structures, in the asymptotic sense.

#### 3.1 Constant Query Time

The query time of  $O(2^{id})$  is constant with respect to  $n$  for any  $i$  and  $d$ . Thus, it is possible to achieve very good update performance while still maintaining constant lookup time. We will first present the general idea and then continue with a detailed presentation of the two dimensional case followed by the general case.

The algorithm splits each dimension in  $\sqrt[n]{n}$  blocks (each with side-length  $\sqrt[n]{n}$ ). Consider Fig. 4. We make the important observation that it is not only possible

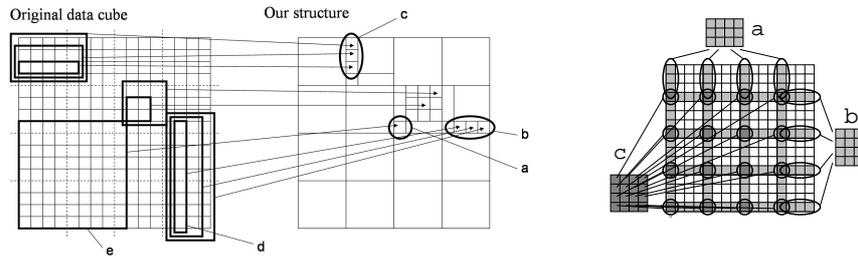


Fig. 3. 2-dimensional example

Fig. 4. Recursion in a 2 dimensions

to apply the algorithm recursively *within* each block, but that all border regions of  $k$  dimensions together with the corresponding border regions from other cells in  $d - k$  dimensions form a subproblem of size  $O(\sqrt[n]{n})$  and dimensionality  $d$ . See Fig. 4 for a 2-dim example of the recursion technique. Here, one row of 1-dim. subproblems (a) form a new 2-dim. subproblem. In the same way, one column of 1-dim. subproblems (b) form a new 2-dim. subproblem. Additionally, the first element of all blocks (0-dim) form a new 2-dim. subproblem (c). Thus, we only have to recurse over subproblems of the same dimensionality. In this way, we can not only describe our fast algorithm simply, but perform the complexity analysis of the algorithm neatly.

Consider the one-dim. example of Figure 2 again. At just one recursion level, our structure would be the same as the SRPS. However, at two recursion levels, the structure would be formed in each of the inner regions as well (in order from left: [3, 3, 5], [5, 9, 10], [3, 3, 5] and [3, 3, 5]). Unfortunately, it takes a much larger example to actually form a two-level structure. Each border region (in order from left: [1], [7], [27] and [32]) together form a subproblem. The structure is recursively applied to them as well. This would yield the vector [1, 6, 27, 5]. This vector would then be stored instead of the border elements. The structure is applied to the pairwise difference of the elements ([1, 6, 20, 5]), not the elements themselves. Unfortunately, the inner regions in this very small example becomes only one element (the 6 and the 5).

Let  $d$  be the dimension of the data cube and  $n_i, 0 \leq i \leq d-1$  be the size of the data cube for dimension  $i$ . Furthermore, let  $e_{(x_0, x_1, \dots, x_{d-1})}$  denote the elements of the original data cube, where  $0 \leq x_i \leq n_i - 1$ . Similarly, let  $g_{(x_0, x_1, \dots, x_{d-1})}$  denote the elements of our data structure. Next, we will present the data structure in detail. First, a two-dimensional structure is presented, where after the general case is introduced.

Consider a single block  $b_{(x,y)} = g_{(k_0 \lceil \sqrt{n_0} \rceil + x, k_1 \lceil \sqrt{n_1} \rceil + y)}$ ,  $x \in [0, \lceil \sqrt{n_0} \rceil - 1]$ ,  $y \in [0, \lceil \sqrt{n_1} \rceil - 1]$  for any  $k_0$  and  $k_1$  (such that the block is within the data cube) of an  $n_0$  by  $n_1$  structure of two dimensions. Then  $b_{(0,0)}$  (marked “a” in Fig. 3) will contain the sum of all elements of the original data cube that has smaller index than both  $x$  and  $y$  (the elements to the lower left, marked “e”, in Fig. 3). Next, consider the elements  $\{b_{(x,0)} : x \in [1, \lceil \sqrt{n_1} \rceil - 1]\}$  (marked “b” in Fig. 4). The element  $b_{(1,0)}$  will contain the sum of all elements from the original data cube with smaller y-index than  $b_{(1,0)}$ , but with the same x-index. This is shown with “d” in Fig. 3. Observe that it is not the elements of the structure that is summed, but the elements of the original data cube. In the same way,  $b_{(2,0)}$  contains the sum of the elements with smaller y-index, but with  $x = 2$  plus  $b_{(1,0)}$ . This holds in general. The fact that  $b_{(x,0)} = b_{(x-1,0)} + e_{(k_0 \lceil \sqrt{n_0} \rceil + x, k_1 \lceil \sqrt{n_1} \rceil)}$  makes the elements a prefix sum. The elements  $\{b_{(0,y)} : y \in [1, \lceil \sqrt{n_0} \rceil - 1]\}$  (marked “c” in Fig. 3) stores elements in the same way, but with coordinates swapped.

We observe that each block in the two-dimensional case contains one prefix sum of two dimensions (white in Fig. 4), two prefix sums of one dimension (light grey in Fig. 4) and one prefix sum of zero dimension (a scalar, dark grey in Fig. 4). In general, for  $d$  dimensions, each block will contain  $\binom{d}{k}$  prefix sums of  $k$  dimensions. This can be realized from an algebraic point of view. The elements for the prefix sums within a single block can be obtained by fixing selected dimensions to 0 (relative to the block) and letting the other dimensions vary to span the region of the prefix sum. If we choose to fix  $k$  of the dimensions, it can be done in  $\binom{d}{k}$  ways.

Consider an arbitrary block

$$b_{(i_0, i_1, \dots, i_{d-1})} = g_{(k_0 \lceil \sqrt{n_0} \rceil + i_0, k_1 \lceil \sqrt{n_1} \rceil + i_1, \dots, k_{d-1} \lceil \sqrt{n_{d-1}} \rceil + i_{d-1})}$$

of our structure, where  $k_0, k_1, \dots, k_{d-1}$  selects block. Then we have that

$$b_{(i_0, i_1, \dots, i_{d-1})} = \sum_{j_0=l_0}^{h_0} \sum_{j_1=l_1}^{h_1} \dots \sum_{j_{d-1}=l_{d-1}}^{h_{d-1}} e_{(j_0, j_1, \dots, j_{d-1})}$$

where  $i_j = 0 \Leftrightarrow l_j = 0, h_j = k_j \lceil \sqrt{n_j} \rceil$   
and  $i_j > 0 \Leftrightarrow l_j = k_j \lceil \sqrt{n_j} \rceil + 1, h_j = k_j \lceil \sqrt{n_j} \rceil + i_j$  for  $j \in [0, d-1]$   
iff  $\exists i \in \{i_0, i_1, \dots, i_{d-1}\} : i = 0$  .

The last condition restricts the validity of the formula to the borders of the block (where at least one index is zero). For the rest of the block (the local prefix sum, no indices are zero) we have that

$$b_{(i_0, i_1, \dots, i_{d-1})} = \sum_{j_0=a_0+1}^{a_0+i_0} \sum_{j_1=a_1+1}^{a_1+i_1} \dots \sum_{j_{d-1}=a_{d-1}+1}^{a_{d-1}+i_{d-1}} e_{(j_0, j_1, \dots, j_{d-1})}$$

for  $i_j \in [1, \lceil \sqrt{n_j} \rceil - 1], a_j = k_j \lceil \sqrt{n_j} \rceil, j \in [0, d-1]$  .

The above discussed structure is a one-level recursive structure ( $i = 1$ , see analysis).

Now, we present the recursion for the general case. Consider Fig. 4. Within each block (the inner region), the algorithm can be applied recursively. Consider the prefix sum (white elements) of the inner region of a block. The algorithm could be applied to the original data cube that corresponds to the prefix sum (the pairwise difference). Consider element  $(0, 0)$  of *all* blocks (the dark grey elements). They also represent a prefix sum and GRPS can be applied recursively to their pairwise difference. To see this, remember that each of those elements contains the sum of all elements from  $e_{(0,0)}$  up to  $b_{(0,0)}$  (in two dimensions). Thus, element  $(0, 0)$  from *all* blocks together forms a 2-dimensional prefix sum. Similar reasoning can be applied to elements  $\{b_{(x,0)} : x \in [1, \lceil \sqrt{n_0} \rceil]\}$  and  $\{b_{(0,y)} : y \in [1, \lceil \sqrt{n_1} \rceil]\}$  (light grey in Fig. 3). One row (or one column) of such elements form a two dimensional prefix sum and the GRPS can be applied recursively to those sums.

Instead of storing the elements of the structure directly, we store the elements of the recursive application of the algorithm. This is possible since the structure does not require extra space, and therefore, a recursive application does not increase the space requirement. Observe that the size of each dimension of the subproblems is  $O(\sqrt{n_i})$ . In general, the subproblems consists of the elements from the data cube that can be obtained by varying the index *within* blocks for certain dimensions and varying *the block* for the other dimensions. As an example, we show a single box for our structure in 3 dimensions in Fig. 5.

For an update, on a one-level structure, it is necessary to rebuild one block and all of the affected border regions (regions that include the updated element in their sum). However, for a multi-level recursive structure, it is only necessary to update the smaller block in the recursive application and the smaller border regions. A query can be processed fairly simple: The prefix sum (or the

range-sum) can be computed by following the recursive structure and adding the appropriate elements.

With the above compact recursive structure, our algorithm achieve the constant time range sum queries.

**Theorem 1.** *The algorithm, described above, has a range-sum query complexity of  $O(2^{id})$ , where  $i$  is the recursion depth and  $d$  is the dimension of the data cube.*

*Proof.* Let  $T_d(n, i)$  denote the query complexity. Clearly,

$$T_d(n, i) = \begin{cases} \sum_{j=0}^d \binom{d}{j} T_d(\sqrt{n}, i-1) & , \text{if } i > 0 \\ 1 & , \text{if } i = 0 \end{cases} .$$

For  $i > 0$ , we have

$$T_d(n, i) = \sum_{k=0}^d \binom{d}{k} T_d(\sqrt{n}, i-1) = 2^d T_d(\sqrt{n}, i-1) = 2^{id} .$$

At the same time, our algorithm requires sub-linear time for update operations while maintaining constant range-sum queries.

**Theorem 2.** *The algorithm, described above, has an update complexity of  $O(2^{di} (\sqrt[i]{n})^d)$ , where  $i$  is the recursion depth and  $d$  is the dimension of the data cube.*

*Proof.* Let  $U_d(n, i)$  denote the update time. Hence, from the description of the algorithm we have,  $U_d(n, i) = 2U_d(\sqrt{n}, i-1)$ , if  $i > 0$ , and  $U_d(n, i) = n^d$ , if  $i = 0$ . For  $i > 0$ , we have  $U_d(n, i) = \sum_{k=0}^d \binom{d}{k} U_d(\sqrt{n}, i-1) = 2^d T_d(\sqrt{n}, i-1) = 2^{id} (\sqrt[i]{n})^d$ .

### 3.2 Update-Query Time Trade-Off

Notice that the time complexity of range sum query and of update depends not only on the data size  $n^d$ , but also on the number of recursion levels,  $i$ , of the data structure (parameter). By choosing the parameter  $i$ , we obtain different trade-off between the running time of queries and updates. Hence,  $i$  can be chosen to tailor the performance of the structure to the needs of the user. If queries are much more common than updates, probably a constant query time is desirable. If  $i = O(1)$ , then the query complexity is always constant (with respect to  $n$ ), but the update complexity is  $O(\sqrt[i]{n^d})$  with respect to  $n$ .

If, on the other hand, the maximum recursion depth possible, is chosen, we get a query complexity of  $O(\log^d n)$  with respect to  $n$  and an update complexity of  $O(\log^d n)$  with respect to  $n$ , which is the same as the complexity of the SDDC [9]. Therefore,

**Corollary 1.** *If the trade-off parameter,  $i = O(\log \log n)$ , both the update and range-sum complexity becomes  $O(\log^d n)$  (This matches the SDDC). If the trade-off parameter  $i = l$ , for some constant  $l$ , the range-sum complexity is  $O(1)$  and the update complexity is  $O(\sqrt[l]{n^d}) = o(n^d)$ .*

### 3.3 Comparison to previous structures

Our structure has better asymptotic query-update trade-off compared to previous structures. To our best knowledge, the best previously known constant-query-time structure that does not require extra memory is the SRPS that has an update complexity of  $O(\sqrt{n^d})$ . For constant query complexity, our structure has  $O(\sqrt[d]{n^d})$  update complexity. The SDDC, with our proposed trade-off has a query complexity of  $O(n^d)$ , for constant query complexity. The SDDC was designed with query and update complexities of  $O(\log^d n)$ . This can be, asymptotically, matched by our structure by choosing the trade-off parameter,  $i = \log \log n$ .

Additionally, our structure will be easier to parallelize, because the data cube is divided by  $O(\sqrt[n]{n})$  instead of blocks of constant size.

## 4 Analysis of SDDC

In this section, we will present an analysis of the SDDC algorithm [9]. Furthermore, we will introduce a trade-off parameter for the SDDC, which is not introduced in the original SDDC. It is important to observe that the trade-off parameter is the recursion depth in both our structure and the SDDC. Our structure, however, requires a lower recursion depth to achieve the same performance.

**Theorem 3.** *The range-sum complexity of the SDDC is  $\Omega(i^d)$ , where  $i$  is the recursion depth and  $d$  is the dimension of the data cube.*

*Proof.* Let  $T_d(n, i)$  denote the range-sum complexity of SDDC. Since the SDDC algorithm reduce the problem size by a factor of  $k$  after each recursion step, we have  $T_d(n, i) = \sum_{j=0}^d \binom{d}{j} T_j(n/k, i-1)$ , if  $i > 0$  and  $d > 0$  and  $T_d(n, i) = 1$ , if  $d = 0$  or if  $i = 0$ . We prove that  $T_d(n, i) \geq i^d$  by induction on  $d$ . For  $d = 1$ , we know that  $T_1(n, i) = i + 1 \geq i$ .

Assume that  $\forall d' < d : T_{d'}(n, i) \geq i^{d'}$ . Hence,  $T_d(n, i) = \sum_{j=0}^d \binom{d}{j} T_j(n/k, i-1) \geq \sum_{j=0}^{d-1} \binom{d}{j} (i-1)^j + T_d(n/k, i-1) = (i-1+1)^d - (i-1)^d + T_d(n/k, i-1) = i^d + (i-i)^d + T_d(n/k^i, i-i) = i^d + 1 \geq i^d$

The update time of the SDDC algorithm is as follows.

**Theorem 4.** *The update complexity of the SDDC is  $\Omega(n^d/k^{di} + k^d i^d)$ , where  $i$  is the recursion depth and  $d$  is the dimension of the data cube.*

*Proof.* Let  $T_d(n, i)$  denote the update complexity of SDDC. Since the SDDC algorithm reduce the problem size by a factor of  $k$  after each recursion step, we have  $T_d(n, i) = \sum_{j=0}^d \binom{d}{j} T_j(n/k, i-1) k^{d-j}$ , if  $i > 0$  and  $d > 0$  and  $T_d(n, i) = n^d$ , if  $i = 0$ . We prove that  $T_d(n, i) \geq (n/k^i)^d + (ki)^d$  by induction on  $d$ . For  $d = 1$ , we have  $T_1(n, i) = k + T_1(n/k, i-1) = 2k + T_1(n/k^2, i-2) = ik + \frac{n}{k^i}$ . Assume that  $\forall d' < d : T_{d'}(n, i) \geq (n/k^i)^{d'} + (ki)^{d'}$ . Hence,  $T_d(n, i) = \sum_{j=0}^d \binom{d}{j} T_j(n/k, i-1) k^{d-j}$

$1)k^{d-j} = \sum_{j=0}^{d-1} \binom{d}{j} T_j(n/k, i-1)k^{d-j} + T_d(n/k, i-1)$ . Consider the first term in this equation. We have

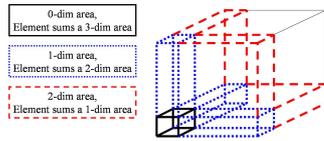
$$\begin{aligned} \sum_{j=0}^{d-1} \binom{d}{j} T_j(n/k, i-1)k^{d-j} &\geq \sum_{j=0}^{d-1} \binom{d}{j} \left( \left( \frac{n}{k^{i-1}} \right)^j + k^j (i-1)^j \right) k^{d-j} = \\ &= k^d \sum_{j=0}^{d-1} \binom{d}{j} \left( \left( \frac{n}{k^i} \right)^j (i-1)^j \right) \\ &= k^d \left( 1 + \frac{n}{k^i} \right)^d - \left( \frac{n}{k^{i-1}} \right)^d + k^d (1+i-1)^d - k^d (i-1)^d = \\ &\left( k + \frac{n}{k^{i-1}} \right)^d - \left( \frac{n}{k^{i-1}} \right)^d + k^d i^d - k^d (i-1)^d \geq k^d i^d - k^d (i-1)^d . \end{aligned}$$

By telescoping, we obtain  $T_d(n, i) \geq k^d(i^d - (i-1)^d) + T_d\left(\frac{n}{k^i}, i-i\right) = k^d i^d + \left(\frac{n}{k^i}\right)^d$ .

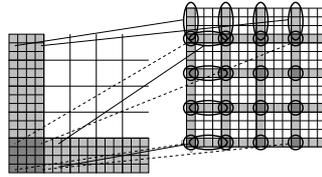
## 5 Further Improvements

The memory access of our structure (and other similar structures) are quite scattered, which could lead to poor performance on cache systems or on secondary storage.

We can overcome most of this problem by reorganizing the memory layout of the algorithm. In our structure, we suggest storing all the border cells from blocks associated with one recursion call in the same place. This is automatically performed for the inner cells (since they already are clustered into the inner cells). Instead of storing the border cells in their respective block, we suggest storing the border cells first (at low index) and then storing all the inner cells in blocks that are one element smaller than before (because of the missing border cells). This should be performed for each recursion level. This way, the data access becomes more concentrated to the same region. See Figure 6 for a two dimensional example.



**Fig. 5.** Single box of 3-dimensional structure



**Fig. 6.** Efficient space localization

## 6 Conclusions

We have presented a space-efficient, range-sum data structure and algorithm for use on data cubes. The query performance of the structure is  $O(2^{id})$  while the update performance is  $O((2^i \sqrt[i]{n})^d)$ . This provides a better query-update trade-off than previously known structures. The trade-off can easily be tuned via the trade-off parameter  $i$ .

We also presented a way to improve the space-localization of the structure, so that access to the data cube becomes less scattered.

It would be interesting to investigate if it is possible to achieve a better update-query trade-off. It would also be interesting to investigate the existence of a structure with good space-time trade-off.

## References

1. Ho, C.T., Bruck, J., Agrawal, R.: Partial sum queries in olap data cubes using covering codes. *IEEE Transactions on Computers* **47** (1998) 1326–1340
2. Codd, E.: Providing OLAP (on-line analytical processing) to user-analysts: an it mandate. Technical report, E.F. Codd and Associates (1993)
3. Gray, J., Bosworth, A., Layman, A., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In: *Proceedings of 12th International Conference on Data Engineering*. (1996) 152–159
4. Li, H.G., Ling, T.W., Lee, S.Y.: Range-max/min query in OLAP data cube. In: *DEXA 2000*. Volume 1873 of *LNCS*. (2000) 467–476
5. Li, H.G., Ling, T.W., Lee, S.Y.: Hierarchical compact cube for range-max queries. In: *Proceedings of the 26th VLDB Conference*. (2000)
6. Chun, S.J., Chung, C.W., Lee, J.H., Lee, S.L.: Dynamic update cube for range-sum queries. In: *Proceedings of the 27th VLDB Conference*. (2001)
7. Chun, S.J., Chung, C.W., Lee, J.H., Lee, S.L.: Space-efficient cubes for OLAP range-sum queries. Technical report, Korean Advanced Institute of Science and Technology (2002)
8. Geffner, S.P., Riedewald, M., Agrawal, D., Abbadi, A.E.: Data cubes in dynamic environments. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (1999)
9. Riedewald, M., Agrawal, D., Abbadi, A.E., Pajarola, R.: Space efficient data cubes for dynamic environments. In: *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWak)*. (2000) 24–33
10. Riedewald, M., Agrawal, D., Abbadi, A.E.: Flexible data cubes for online aggregation. In: *ICDT. LNCS* (2001) 159–173
11. Ho, C., Agrawal, R., Megiddo, N., Srikant, R.: Range queries in OLAP data cubes. In: *Proceedings of the ACM SIGMOD*. (1997) 73–88
12. Poon, C.K.: Orthogonal range queries in OLAP. In: *Proceedings of the 8th International Conference on Database Theory* (2001). (2001) 361–374