Dynamic Update Cube for Range-Sum Queries

Seok-Ju Chun† Chin-Wan Chung‡ Ju-Hong Lee† Seok-Lyong Lee†

†Department of Information and Communication Engineering
‡Department of Computer Science
Korea Advanced Institute of Science and Technology
{chunsj,chungcw,jhlee,sllee}@islab.kaist.ac.kr

Abstract

A range-sum query is very popular and becomes important in finding trends and in discovering relationships between attributes in diverse database applications. It sums over the selected cells of an OLAP data cube where target cells are decided by the specified query ranges. The direct method to access the data cube itself forces too many cells to be accessed, therefore it incurs a severe overhead. The response time is very crucial for OLAP applications which need interactions with users. In the recent dynamic enterprise environment, data elements in the cube are frequently changed. The response time is affected in such an environment by the update cost as well as the search cost of the cube.

In this paper, we propose an efficient algorithm to reduce the update cost significantly while maintaining reasonable search efficiency, by using an index structure called the *Δ-tree*. In addition, we propose a hybrid method to provide either an approximate result or a precise one to reduce the overall cost of queries. It is useful for various applications that need a quick approximate answer rather than an accurate one, such as decision support systems.

1. Introduction

On-Line Analytic Processing (OLAP) [Cod93] is a category of database technology that allows analysts to gain insight on an aggregation of data through the access to a variety of possible views of information. It often needs to summarize data at various levels of detail and on various combinations of attributes. Typical OLAP applications include product performance and profitability, effectiveness of a sales program or a marketing campaign, sales forecasting, and capacity planning [BS97]. Among various OLAP application areas, a data model for the multidimensional database (MDDB), which is also known as a data cube[HAMS97], becomes increasingly important.

A data cube is constructed from a subset of attributes in the database. Certain attributes are chosen to be measure attributes, i.e., the attributes whose values are of interest. Other attributes are selected as dimensions or functional attributes [GAES99]. The measure attributes are aggregated according to the dimensions. Consider a data cube maintained by a car-sales company. It is assumed that the data cube has four dimensions MODEL_NO, YEAR, REGION, COLOR, and one measure attribute AMOUNT_OF_SALES. Let the domain of MODEL_NO contain 30 models, of YEAR be from 1990 to 2001, of REGION contain 40 regions, and of COLOR be {white, red, yellow, blue, gray, black}. Then the data cube has 30×12×40×6 cells, and each cell contains AMOUNT_OF_SALES as a measure attribute for the corresponding combination of 4 functional attributes, i.e. MODEL_NO, YEAR, REGION, and COLOR. A data cube provides a useful analysis tool on data called a *range-sum query* that applies an aggregate operation to the measure attribute within the range of the query [GAE00]. A Typical example includes "*Find the total amount of sales in Seoul for all models with red color between 1995 and 2000.*" Queries of this form are very popular and important in OLAP.

It is natural that the response time is very crucial for the OLAP application which needs user-interaction. The direct method to process the range-sum query is to access the data cube itself. But it suffers from the fact that too many cells need to be accessed to get the range-sum. The number of cells to be accessed is proportional to the size of the sub-cube defined by the query. To enhance the

*This work was supported by the Korea Research Foundation Grant (KRF-2000-041-E00262).

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

search efficiency, the prefix sum approach [HAMS97] has been proposed, which uses an additional cube called a prefix sum cube (PC), to store the cumulative sum of data. This method however focuses on reducing the search cost. The current enterprise environment forces data elements in the cube to be dynamically changed. In such an environment, the response time is affected by the update cost as well as the search cost of the cube.

Recently, various excellent studies [GAES99, CI99, LWO00, GAE00] have been made to reduce the update cost. Those methods use additional data structures such as the relative prefix sum cube (RPC) to minimize the update propagation over the prefix sum cube. However, those approaches have some limitation in the context that they still have the update propagation problem even though they have reduced it in some degree, since the RPC is a slight transformation of the PC. Furthermore, their update speed-up is accomplished by the sacrifice of the search efficiency. In many OLAP applications, it becomes an important issue to improve the update performance while minimizing the sacrifice of the search efficiency.

In this paper, we propose an efficient algorithm to make a drastic cut in the update propagation by using an index structure called the Δ-tree. Various multi-dimensional index structures [SRF87, BKSS90, BKK96] have been proposed since the R-trees [Gut84]. The Δ-tree is a modified version of the R*-tree [BKSS90] to store the updated values of a data cube and to support the efficient query processing. Our algorithm takes advantage of the traditional prefix sum approach to gain considerable search efficiency with a significant reduction of the update cost.

Furthermore, in many current enterprise applications like the decision support system, there are a number of trial-and-error steps involved in getting the right answer. It forces range-sum queries to be executed too many times, which causes severe query cost. Thus it is important to provide the facility to get a quick approximate result rather than an accurate one to support the decision making process timely. We propose a hybrid method to provide either an approximate result or a precise one in order to reduce the overall costs of queries for collecting information for decision making.

1.1 Related work

As we introduced briefly in the previous section, various approaches that address the query on an OLAP data cube were proposed. Ho, et el. [HAMS97] have presented an elegant algorithm for computing range queries in data cubes which we call the *prefix sum* approach. The essential idea of the prefix sum approach is to precompute many prefix sums of the data cube, which can be used to answer ad hoc queries at rum-time. This approach turned out to be very powerful. Range-sum queries were processed in constant time regardless of the size of the data cube. But, it is very expensive to maintain the prefix sum cube when data elements in the cube are frequently changed.

To reduce the update propagation in the prefix sum cube, Geffner et el. [GAES99] presented an algorithm for computing range queries in data cubes which they called the *relative prefix sum* approach. They tried to balance the query-update tradeoff between the direct method and the prefix sum approach. This approach is however impractical in the data cube of high dimensions and high capacity since the update cost increases exponentially. Chan and Ioannidis [CI99] proposed a new class of cube representations called Hierarchical Cubes, which was based on two orthogonal dimensions. They have shown that a particular cube design called the Hierarchical Band Cube has a significantly better query and update trade-off than that of the algorithm proposed[GAES99]. But the index mapping from a high-level "abstract" cube to a low-level "concrete" cube is too complicated for implementation. They did not verify the analytical results of their method experimentally.

More recently, Geffner et el. [GAE00] proposed the *Dynamic Data Cube* which was designed by decomposing the prefix sum cube recursively. They assumed that each dimension of a data cube was of the same size, and constituted a tree structure by a decomposition technique. But the data cube of a practical environment, like the example of the car-sales company in the previous section, has the dimensions of different sizes. (In our example, the size of each dimension is 30, 12, 40, and 6, respectively.) Dimensions of different sizes make it difficult to keep the balance of the tree while decomposing the prefix sum cube. Besides, if the data cube is of high dimensions and high capacity, it is difficult to apply their approach since the tree becomes too large, so the approach incurs a high computation overhead.

1.2 Contributions

In this paper, we present a new technique called a dynamic update cube which exploits an index structure, that is, the Δ-tree. Our contributions are summarized as follows:

- We have proposed an efficient algorithm to take advantage of the prefix sum approach and to reduce the update cost significantly using the Δ-tree. We presented the comparison on the update complexities of various methods. The update complexity of our algorithm is O(log N_u), with respect to N_u, the number of changed cells in the data cube. We provided an analysis using various sizes of data cube and experimental evaluation which showed that our method performed very efficiently on various dimensionalities, compared to other methods.
- We have proposed a hybrid method to provide either an approximate result or a precise one with respect to OLAP range-sum queries, and also proposed the method to reduce the approximation error considerably. To our knowledge, the proposed approach is the first work specifically addressing a hybrid method that gives

both approximate and accurate answers at the same time on users' demands. The extensive experiment on the approximation method demonstrates a remarkable speed-up in the query processing while preserving a considerable accuracy

The remainder of the paper is organized as follows: The preliminary information for the prefix sum approach is described in Section 2. Section 3 provides a detailed description of our proposed work. A hybrid method that provides either an approximate result or a precise one is given in Section 4. Experimental results with respect to the performance evaluation of proposed algorithms are presented in Section 5 and we give conclusions in Section 6.

2. Preliminary

In this section, we introduce the background information regarding the prefix sum cube which is closely related to our proposed method. In the prefix sum approach, a prefix sum cube *PC* of the same size as the data cube A, stores various precomputed prefix sums of *A*. Each cell of *PC* contains the sum of all cells up to and including itself in the data cube *A*. Figure 1 shows an 6×8 data cube and its prefix sum cube. Cell *PC*[4,6] contains the sum of all cells in the range *A*[0,0] to *A*[4,6]. The sum of the entire cube *A* is found in the last cell *PC*[5,7].

Let $D = \{1, 2, \ldots, d\}$ denote the set of dimensions and n_i denote the number of cells in dimension i. Ho, et el. [HAMS97] have presented a simple method which needs $N = \prod_{i=1}^{d} n_i$ additional cells to store certain precomputed prefix-sums such that any d-dimensional range-sum can be computed in $2^d - 1$ computation steps, based on up to 2^d appropriate precomputed prefix sums. Formally, for all $0 \le x_i < n_i$ and $i \in D$,

$PC[x_1,x_2,\ldots,x_d] = Sum(0 : x_1, 0 : x_2, \ldots, 0 : x_d) = \sum_{i_1=0}^{x_1} \sum_{i_2=0}^{x_2} \cdots \sum_{i_d=0}^{x_d} A[i_1, i_2, \ldots, i_d]$

For example, When $d = 2$, we precompute, for all $0 \le x < n_1$ and $0 \le y < n_2$,

$PC[x,y] = Sum(0{:}x, 0{:}y) = \sum_{i=0}^{x} \sum_{j=0}^{y} A[i, j]$.

Figure 1 shows an example of $A[x_1,x_2]$ and its corresponding $PC[x_1,x_2]$ for $d = 2$. The prefix sum approach is very powerful. It provides range-sum queries in constant time, regardless of the size of the data cube.

The Lemma 2.1 below provides how any range-sum of A can be computed from up to 2^d appropriate elements of *PC*. The left hand side of the below equation specifies a range-sum of A. The right side of the equation consists of 2^d additive terms, each of which is from an element of *PC* with a sign "+" or "–" determined by the product of all $s(i)$'s . For notational convenience, let $PC[x_1, x_2, \ldots, x_d] = 0$ if $x_j = -1$ for some $j \in D$.

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

(a) data cube *A*

Index	0	1	2	3	4	5	6	7
0	4	9	11	19	22	29	34	40
1	6	12	19	30	40	49	58	66
2	11	20	36	50	64	80	90	101
3	14	28	50	65	87	108	119	136
4	17	33	56	75	104	133	150	171
5	23	41	66	91	121	159	181	204

(b) prefix sum cube *PC*

Figure 1. Example of an 6×8 original data cube *A* and its prefix sum cube *PC*

Lemma 2.1 [HAMS97]. *For all $j \in D$, let*

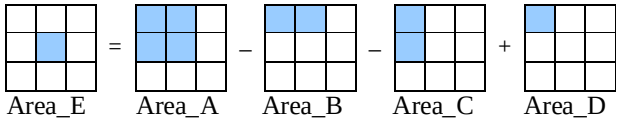
Then, for all $j \in D$, $s(j) = \begin{cases} 1, & \text{if } x_j = h_j, \\ -1, & \text{if } x_j = l_j - 1. \end{cases}$

$Sum(l_1{:}\, h_1, l_2{:}\, h_2, \ldots, l_d{:}\, h_d)$

$= \sum_{\forall x_j \in \{l_j - 1, h_j\}} \left\{ \left(\prod_{i=1}^{d} s(i) \right) * PC[x_1, x_2, \ldots, x_d] \right\}$ ■

Example 2.2 When $d = 2$, the range-sum $Sum(l_1{:}h_1, l_2{:}h_2)$ can be obtained by the computation: $PC[h_1, h_2] - PC[h_1, l_2-1] - PC[l_1-1, h_2] + PC[l_1-1, l_2-1]$. As illustrated in Figure 1, the range-sum $Sum(1 : 4, 2 : 6)$ can be derived from $PC[4,6] - PC[0,6] - PC[4,1] + PC[0,1] = 150 - 34 - 33 + 9 = 102$. ■

Figure 2 gives a geometrical explanation of the computation for a two-dimensional case.



Area_E Area_A Area_B Area_C Area_D

Figure 2. A geometric illustration of the two dimensional case

3. Proposed work

3.1 Motivation

With the advances of the internet technologies such as the World Wide Web, we are able to use diverse applications including OLAP servers regardless of time and location. Users may be widely spread geographically and also a great number of users may want to use a large scale OLAP server concurrently. Performance in these environ-

ments becomes an issue when we support the query processing and dynamic data updates at the same time.

Furthermore, in the previous methods based on the prefix sum cube, many queries of users is blocked if the ranges of these queries include a single cell which must be changed due to the update process. Otherwise, the queries produce incorrect answers. Namely, one update operation can cause many queries to be blocked. Therefore, It is clear that the blocking will degrade the overall performance of the OLAP server when the update cost is high.

The OLAP server is widely used as a system for supporting the decision making process. A number of users may want to issue a large number of queries related to theirs' concerns until they reach some decision. If all queries require precise answers, they incur high cost of their execution and a great load to the OLAP server. On the other hand, if all queries need approximate answers, they also make users be confused with inexact answers. We need some hybrid method that provides approximate answers during the process of focusing and exact answers for queries of interest. However, the previous methods based on the prefixed sum cube do not provide this kind of hybrid method.

3.2 Idea

In a dynamic OLAP environment, cells of the data cube are frequently changed. The problem is that the cost of updating the prefixed sum cube is very high. The basic idea is that we store and manage changed cells using the virtual cube, called the 'dynamic update cube' instead of updating the prefixed sum cube directly. When a range-sum query is processed, the prefixed sum cube and the dynamic update cube are manipulated simultaneously.

Regardless of being dense or sparse of the data cube, the prefixed sum cube is always dense. On the other hand, the dynamic update cube is sparse because it involves only the changed cell of the data cube.

Example 3.1 As shown in Figure 3, the prefixed sum cube(*PC*) is always dense since it stores the cumulative sums of data cube cells even though the data cube is sparse. In this Figure, '×' indicates a changed cell and 'Δ' indicates the difference of the values of a changed cell, that is, $\Delta = \times_{new} - \times_{old}$. Since the dynamic update cube stores Δ values, it is sparse. ■

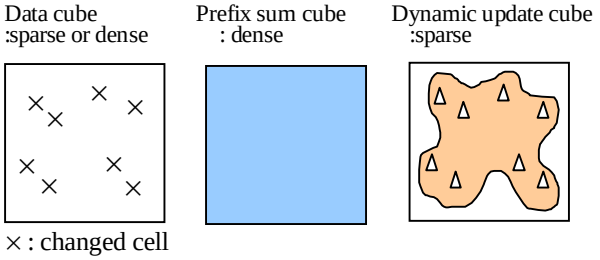


Figure 3. The basic concept of the dynamic update cube

The positions of the dynamic update cube cells can be represented as multidimensional points and so, these cells are stored into the multidimensional index structure, called the 'Δ-Tree'. The cells that are spatially close each other are clustered into a corresponding the minimum bounding rectangle(MBR). When searching the Δ-Tree, non-overlapping MBRs of the Δ-Tree are pruned efficiently. More details are explained in Section 3.4. The positions and Δ values of the dynamic update cube cells are stored into the Δ-Tree.

The idea proposed in this paper are summarized as follows:

- Since the prefixed sum cube is dense and the dynamic update cube is sparse, whenever the data cube changes, we do not update directly to the prefixed sum cube. Instead we store the changes of the data cube into the Δ-Tree and then manage it. This reduces the update cost and resolves the problem of propagating updates in the prefixed sum approach.
- When processing a range-sum query, we can get an approximate answer by searching the Δ-tree partially. That is, searching is performed from the root to an internal node of the level i instead of a leaf node. The details of calculating an approximate answer are explained in Section 4.
- The size of the Δ-tree can increase as the cells of the data cube are changed. When the Δ-tree is too large, the cost of search and update becomes high. Thus, all information stored in the Δ-Tree needs to be reflected on the prefix sum cube(so-called 'bulk updates') periodically, depending on applications, i.e., weekly, monthly, or at some threshold.

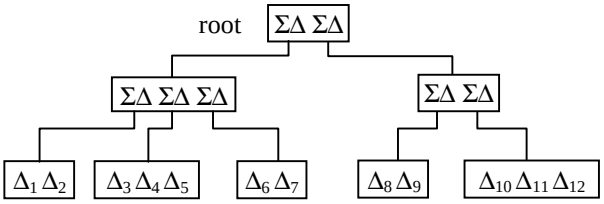


Figure 4. The structure of Δ-tree

3.3 The Δ-tree

In this section, we introduce the structure of the Δ-tree, as shown in Figure 4. The construction process of the Δ-tree is the same as that of the R*-tree. Initially the Δ-tree has only a directory node(called the root node). Whenever the data cube cell is updated, the difference(Δ) between the new and old values of the data cube cell and its spatial position are stored into the Δ-tree. We define the Δ-tree formally as follows:

Definition 3.2 (the Δ-tree)

1. A directory node contains (L_1, L_2,..., L_n), where L_i is the tuple about the *i*'th child node C_i and has the form (ΣΔ, *M*, cp_i, MBR_i). ΣΔ is the sum of all ΣΔ values(Δ values) of C_i when C_i is a directory node(data node). cp_i is the address of C_i and MBR_i is the *MBR* enclosing all entries in C_i . *M*

has the form (μ_1, μ_2, ... , μ_d) where d is the dimension and μ_j is the mean position of the j'th dimension of MBR_i which is defined as follows:

$$\mu_j = \frac{\sum_{m=1}^{n_j} mF_j(m)}{\sum_{m=1}^{n_j} F_j(m)},$$

where $F_j(m) = \sum_{\substack{k_h=1,n_h \\ k_j=m \\ h\neq j}} f(k_1,...,k_j,...,k_d)$,

$f(k_1, k_2,..., k_d)$ is the value of an update position (k_1, k_2,..., k_d) in MBR_i with $1 \le k_j \le n_j$, and n_j is the number of partitions of the j'th dimension of MBR_i.

2. A data node is at the level 0 and it contains (D_1, D_2,..., D_n), where D_i is the tuple about i'th data entry and has the form (P_i, Δ_i). P_i is the position index and Δ_i is the difference of the changed cell.

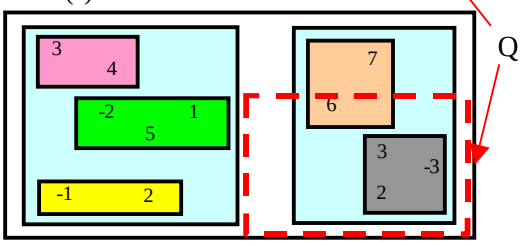
The objective of using $\Sigma\Delta$ is to provide both fast and approximate answers of the range-sum query and the objective of using M is to improve the approximation technique (see example 4.2 in Section 4.2).

3.4 Range-sum query

We use both the prefix sum cube PC and the Δ-tree in order to answer the range-sum query. As we mentioned before, PC includes the information which had been most recently bulk updated while the Δ-tree includes the information which has been updated from then on. The update cells which are spatially close each other are clustered into a corresponding MBR.

Index	0	1	2	3	4	5	6	7
0	7*	5	2	8	3	7	12*	6
1	2	5*	5	3	7	2	4	2
2	5	1*	9	4*	4	13*	1	3
3	3	5	11*	1	8	5	4*	6
4	3	2	1	4	7	8	6	1*
5	5*	2	4*	6	1	9	7*	2

(a) Data cube A



(b) Dynamic update cube U

Figure 5. Example of a range-sum query in the data cube and the dynamic update cube

Example 3.3 For the example in Figure 5, the cells marked by the symbol '*' in the data cube A indicate that they have been updated from the data cube in Figure 1-(a). In Figure 5-(b), each MBRs in the lowest level of the dynamic update cube contain these cells which so far have not been reflected to PC. ■

When a range-sum query Q, where Q is (l_1: h_1, l_2: h_2,..., l_d: h_d), is given, we use PC and the Δ-tree for obtaining the answer of Q. Let $Sum(Q)$ be a function that returns the answer of Q, $PC\_sum(Q)$ be a function that returns the answer which is calculated from PC, and $\Delta\_sum(Q)$ be a function that return the answer which was found from the Δ-tree . Then, the answer will be:

$Sum(Q) = PC\_sum(Q) + \Delta\_sum(Q)$

Example 3.4 For Figure 5, when a range-sum query Q is given as below, we can obtain the answer of Q using PC and the Δ-tree.

Range-sum query(Q): Select Sum(A.sales)
From A
Where 2 ≤ A.x ≤ 5 and 4 ≤ A.y ≤ 7

We have the answer from the above equation. That is, $Sum(2:5, 4:7) = PC\_sum(2:5, 4:7) + \Delta\_sum(2:5, 4:7)$. The function $PC\_sum(2:5, 4:7)$ can be obtained 'on-the fly' by Lemma 2.1. ■

Definition 3.5 (*Disjoint, Inclusive, Intersecting*)
Let MBR_Q and MBR_T be the MBR of a query Q and the MBR of a node T. The relationship between MBR_Q and MBR_T may formally be defined as
(1) Disjoint iff $MBR_Q \cap MBR_T = \phi$.
(2) Inclusive iff $MBR_Q \supseteq MBR_T$.
(3) Intersecting iff $MBR_Q \cap MBR_T \neq \phi$ and not inclusive.
Note that $MBR_Q \subset MBR_T$ is defined to be intersecting.

When the Δ-tree is traversed to find the result from the function $\Delta\_sum(Q)$, a root node is at first visited, and each entry of the root node is evaluated with respect to the spatial relationship between MBR_Q and MBR_T, as described in Definition 3.5. The brief algorithm of the function $\Delta\_sum(Q)$ is shown as follows:

Algorithm $\Delta\_sum()$
input: query Q, *Δ-tree*
output: *answer*
procedure:
1. Visit the nodes in the Δ-tree in the depth first order starting from the root node. If there is no more node to be visited, return *answer*
2. Each entry of the node is evaluated. There are three cases of the relationship between MBR_Q and MBR_T of each entry, as described in Definition 3.5. Those cases and the corresponding pruning strategies are as follows:
case 1 (*disjoint*):
The entry related to MBR_T is irrelevant to the query Q. Thus, the sub-tree under is pruned.
case 2 (*inclusive*):
$\Sigma\Delta$ of the entry related to MBR_T is added to *answer*. It is not necessary to traverse the sub-tree under the entry any more since $\Sigma\Delta$ is an exact answer with respect to this entry.
case 3 (*intersecting*):

In this case, we have two choices for the range-sum: precise and approximate. To get a precise answer, we need to evaluate every child MBR which is included in MBR_T. To get a approximate answer, we compute the approximate ΣΔ of the entry related to MBR_T, and add it to *answer*. The detail description on how to get the approximate answer is discussed in Section 4. The algorithm *Δ_sum*() is recursively called with the *Δ-tree* replaced by the sub-tree under this entry.

Example 3.6 As shown in Figure 6, when a range-sum query *Q*(2:5,4:7) (dotted box) is given, MBR_1 is disjoint, MBR_2 is intersecting, and MBR_3 is inclusive. Therefore, we can find that the answer of the function *Δ_sum*(2:5, 4:7) is 8. Thus, we complete the function *Sum*(2:5,4:7) as follow:

Sum(2:5,4:7)=*PC*[5,7]–*PC*[1,7]–*PC*[5,3]+*PC*[1,3]+
Δ_sum(2:5, 4:7)= (204 – 66 – 91 + 30) + (6 + 2) = 85. ■

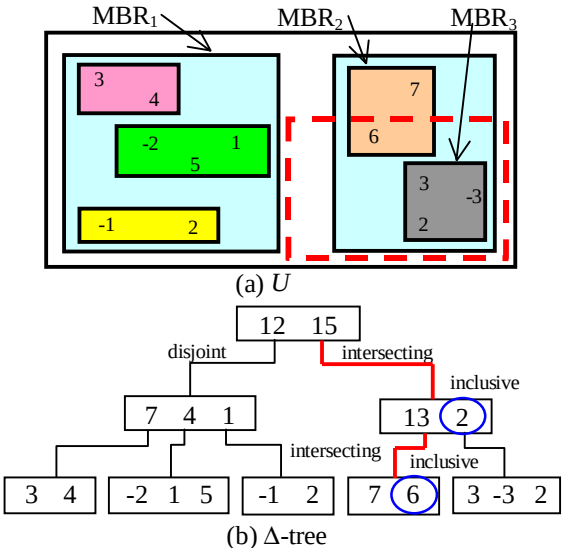


Figure 6. Dynamic update cube U and Δ-tree corresponding to U.

3.5 Updates

When the value of a cell in the data cube is changed, it does not affect the prefix sum cube directly. Instead, we only need to change the value of an appropriate location in the Δ-tree. Let us consider how the update on the value of a cell affects the Δ-tree. The update request is given in the form (*P*, Δ) where *P* is the position index and Δ is the difference from the old Δ value of the changed cell. The first step for the update is to identify the sub-tree into which the update is made. Choosing the target sub-tree for the update is the same as that for R*-tree. By identifying the sub-trees repeatedly, the target data node to reflect the update request is finally chosen. Once the target data node is identified, our method checks whether the data entry with the position *P* exists in the node or not. There are two cases as follows:

Case 1. (when the position *P* exists)
In this case, the update is made in the data entry (P, Δ_{OLD}) where Δ_{OLD} is the existing Δ value of *P*. The Δ value is added to Δ_{OLD}. And then, for all ancestors of the data node in the tree, we set: $(\Sigma\Delta)_{ancestor} = (\Sigma\Delta)_{ancestor} + \Delta$, where $(\Sigma\Delta)_{ancestor}$ is ΣΔ of an ancestor node of the data node. This process is repeated up to the root node.

Case 2. (when the position *P* does not exist)
If the position *P* does not exist, the data entry (*P*, Δ) is inserted in the end of the node. And also, for all ancestors of the data node in the tree, we set: $(\Sigma\Delta)_{ancestor} = (\Sigma\Delta)_{ancestor} + \Delta$, as described in the case 1. Sometimes, an overflow occurs in a node during the insertion process when the number of data entries exceeds a specified threshold. In this case, the node is split into two nodes. We have adopted the same splitting strategy as that of the R*-tree. For more details on insert/split algorithms of the R*-tree, refer to [BKSS90]. Let us assume that the node *S* is split into S_1 and S_2. Then it needs to recalculate ΣΔ's of the parent of S_1 and S_2. Adjusted ΣΔ's of the parent of S_1 and S_2 are reflected to all ancestor nodes in the tree, up to the root node.

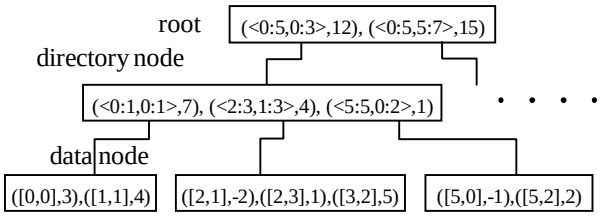


Figure 7. The detail structure of the Δ-tree in Figure 6-(b).

Example 3.7 Let us consider the update process when the value of the cell A[2,3] in Figure 5-(a) is changed from 4 to 6. This process corresponds to the update request ([2,3], 2) into the Δ-tree. As shown in Figure 7, an entry (<0:5,0:3>, 12) is at first selected in the root node. Traversing down the tree, an entry (<2:3,1:3>, 4) is chosen in the intermediate node. Since the data entry with the position [2,3] is found in the node which is pointed to by the entry (<2:3,1:3>, 4), the Δ value 2 is added to the old value 1, resulting in 3. After changing the data node, the ΣΔ's of ancestor nodes are changed from (<2:3,1:3>, 4) and (<0:5,0:3>, 12) to (<2:3,1:3>, 6) and (<0:5,0,3>, 14) respectively. ■

3.6 Time complexity of the dynamic update cube

The dynamic update cube provides a significant efficiency compared to the previous methods such as [HAMS97, GAES99, LWO00, GAE00]. The time complexity of our method for updating a single cell in the Δ-tree is $O(\log N_u)$, where N_u is the number of changed cells. It is usual that the number of changed cells is very small compared to the total size of the data cube. The complexity $O(\log N_u)$

corresponds to the complexity of descending a single path in the tree. Table 1 shows the comparison of complexities among different methods. Here, we assume that $N = n^d$ and *n* is the number of cells in each dimension. Thus, it is clear that our method outperforms other methods since the size of the dynamic update cube is very small than that of the original data cube. that is, $N_u << N$.

Table 1. Time complexities among different methods

Method	Update time
Prefix Sum[HAMS97]	$O(n^d)$
Relative Prefix Sum[GAES99]	$O(n^{d/2})$
Dynamic Data Cube[GAE00]	$O(\log^d n)$
Dynamic Update Cube	$O(\log N_u)$

As an example of the comparison of time complexities shown in Table 1, Table 2 shows the number comparison of the update costs for various methods when the dimensionalities (d in Table) are 2, 4, and 8, and the size (n in Table 1) of each dimension is 10^1 and 10^2. For instance, when d = 4 and $n = 10^2$, the total size of a data cube $N = n^d$ is 10^8. We assume that the fan-out of the Δ-tree is 10, that is, the base of *log* in the complexity of our method is 10. We also used 10 as the base of log for the dynamic data cube. We consider that generally N_u is around 1% of N. Therefore, our method is evaluated for three cases: N_u = 0.1%, 1%, and 10% of N. As we observe the results in Table 2, our method outperforms other methods.

Table 2. The number comparison of update costs for various methods

n	d	$N= n^d$	Prefix-Sum	Relative PS	Dynamic Data Cube	Dynamic Update Cube		
						N_u= 0.001N	N_u= 0.01N	N_u= 0.1N
10	2	10^2	10^2	10^1	11	-	1	1
	4	10^4	10^4	10^2	118	1	2	3
	8	10^8	10^8	10^4	14064	5	6	7
100	2	10^4	10^4	10^2	43	1	2	3
	4	10^8	10^8	10^4	1897	5	6	7
	8	10^{16}	10^{16}	10^8	3600406	13	14	15

The previous methods based on the prefix sum cube compromise the query cost in order to improve the update cost. Our method has a very efficient update performance using the Δ- tree, but it requires for processing queries, therefore, both the previous methods and our method incur additional overhead compared with the prefix sum approach. The experiments in Section 5 demonstrate that the query processing of our method is quite efficient.

4. Hybrid Method

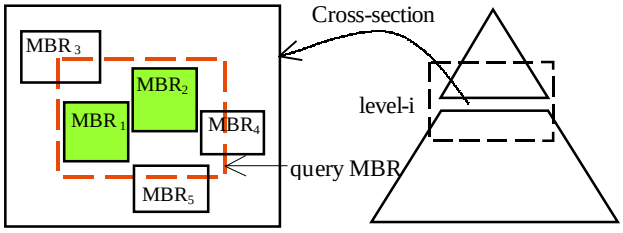
In a real OLAP environment users typically search for trends, patterns, or unusual data behaviors by issuing queries interactively. Thus, users may be satisfied with approximate answers for queries if the response time can be greatly reduced. In this section, we propose a hybrid method to provide either an approximate result or a precise one to reduce the overall cost of queries. It is highly beneficial for various applications that need quick approximate answers rather than time consuming accurate ones, such as decision support systems. We provide the approxim-ation technique and illustrate how to reduce the errors of the approximation technique with a little additional cost.

4.1 Approximation technique

When processing a range-sum query, we can obtain an approximate answer by searching the Δ-tree partially. That is, searching is performed from the root to an internal node of the level i instead of a leaf node. There exist several MBRs which are participated in answering a range-sum query. We can classify these MBRs into two groups as follows:

1. Inclusive MBRs: MBR_i (i=1,…,m), where m is the number of inclusive MBRs.
2. Intersecting MBRs: MBR_j (j=m+1,…,n), where n – m is the number of intersecting MBRs.

Example 4.1 As shown in Figure 8, we can see the level i-th cross-section of the Δ-tree. That is, MBR_1 and MBR_2 are inclusive MBRs, the other side, MBR_3, MBR_4 and MBR_5 are intersecting MBRs. ■



(a) MBRs at the level i of Δ-tree (b) Δ-tree

Figure 8. The shape of query MBR and MBRs in the level i of the Δ-tree.

Let $(\Sigma\Delta)_i$ (i=1,…,m) be the ΣΔ value of the i'th inclusive MBR, and $(\Sigma\Delta)_j$ (j=m+1,…,n) be the ΣΔ value of each intersecting MBR. The answer of the range-sum query at the level i of the Δ-tree can be approximated by the following equation:

Approx_sum(Q) =

$$\sum_{i=1}^{m} (\Sigma\Delta)_i + \sum_{j=m+1}^{n} \left(\frac{Vol\left(MBR_j \cap MBR_Q\right)}{Vol\left(MBR_j\right)} \times (\Sigma\Delta)_j \right) + PC\_sum\ (Q)$$

4.2 Improving approximation technique

In this section, we propose to use the list of mean positions (*M* in Definition 3.2) for improving the approximation technique. We resize the area of a range query for a more accurate approximation and calculate the difference between the approximation value without resizing and that with resizing to find nodes for further searching. Example 4.2 illustrates this.

Example 4.2 As shown in Figure 9, the overlapping region is $(0{:}x, y{:}L_2)$ and μ_1, μ_2 have been calculated. Then

the region is resized for a more accurate approximation. Let us consider the vertical side of the node. [0, μ_1] contains a half of the values. We want to find α such that [0,α] , when values are uniformly distributed, contains the values contained in [0,x] when μ_1 is the mean position. Then $\frac{x-\mu_1}{L_1-\mu_1} = \frac{\alpha - 0.5L_1}{0.5L_1}$. That is, $\alpha = \frac{L_1}{2}\left(1 + \frac{x-\mu_1}{L_1-\mu_1}\right)$. Therefore, 0:$x$ is resized to 0: $\frac{L_1}{2}\left(1 + \frac{x-\mu_1}{L_1-\mu_1}\right)$. The resizing of y:L_2 can be similarly calculated to be $\frac{L_2}{2} \times \frac{y}{\mu_2}$:$L_2$. And, we calculate the difference between the approximation value with resizing and that without resizing to find nodes to be searched down in the next lower level. We select the nodes whose difference values are bigger than others. ■

When searching intersecting nodes in the level i, the error can be reduced much with a little additional overhead if we search down to the level i-1 for a few nodes in the level i having big differences.

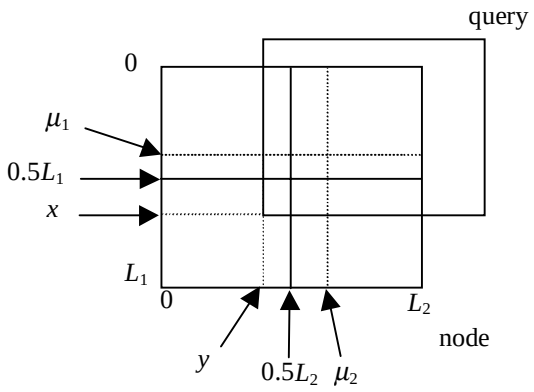


Figure 9. Resizing of the query

5. Experimental Evaluation

In this section, we present the experimental environment and the performance evaluation of our proposed method. The method proposed in this paper uses both the prefix sum cube and the Δ-tree. That is, the update is made in real-time on the Δ-tree and all updates are reflected in the prefix sum cube periodically. Therefore we evaluated the update performance by using the update on the Δ-tree. As for the query efficiency, we considered the prefix sum cube as well as the Δ-tree since the query is processed on both. We evaluated the accuracy of approximate results by accessing both the prefix sum cube and the Δ-tree.

The Δ-tree was implemented by modifying the R*-tree to accommodate ΣΔ and M, and its node size was adjusted to have a reasonable depth (say, 5 or 6) for evaluating approximate results for the hybrid method. Test data sets were generated to have two types of distributions: uniform and Zipf distributions. The z parameter of Zipf distribution was determined to have a constant value (z=0.9) regardless of the dimension. The dimensionalities of the test data are 2, 3, 4, and 5. The cardinality of each dimension d is 1024 for d = 2, 512 for d = 3, 128 for d = 4, and 64 for d = 5, respectively. The number of data elements that are to be inserted into the Δ-tree is 10000 through 50000. Three types of queries are used based on query size (i.e., query volume / data cube volume) as follows: large(=0.1), medium(=0.05), small(=0.01).

All experiments have been done on a Sun Ultra II workstation with 256M main memory and 10G hard disk. The error rates in the experimental results indicate the percentage error. Each experimental result has been obtained by issuing 30 queries and 30 updates for evaluating the range-sum query and the update process respectively, and by averaging the results of them.

Figure 10 shows the efficiency of range-sum queries in the dimensionality of 3 (d = 3) for various query sizes, i.e. large, medium, and small sizes. Figure 11 illustrates the results of the query execution for the medium-size query with respect to the dimensionalities of 3, 4, and 5, respectively. Those results have been obtained by visiting nodes of the Δ-tree from the root to the level 0, i.e. the data node, and thus those results are exact (not approximate). The efficiency of queries is measured by the execution time in second. For d = 3, the result shows a considerable efficiency, that is, the execution time is below one second.

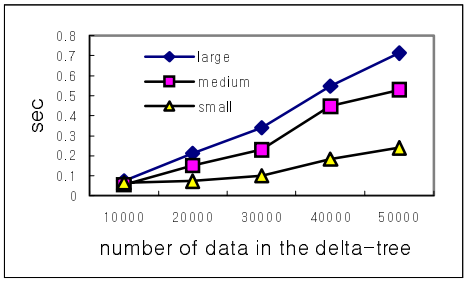


Figure 10. Exact query performance(=level 0) for uniform distribution, dim=3 and query sizes = large,medium,small

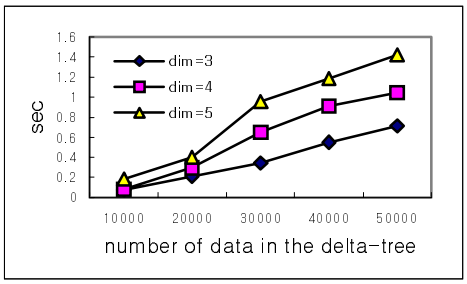


Figure 11. Exact query performance (=level 0) for uniform distribution, dim=3,4,5 and medium size query

Figure 12 shows the performance as the number of page accesses for inserting the value of a changed cell to the Δ-tree. For this experiment, we inserted up to 100000 data elements into the Δ-tree. The X-axis represents the

number of data elements in the Δ-tree while the Y-axis represents the number of page accesses to insert a single value into the tree which corresponds to the depth of the tree. As we observe in the Figure, the number of page accesses is O(log N_u), where N_u is the number of changed cells.

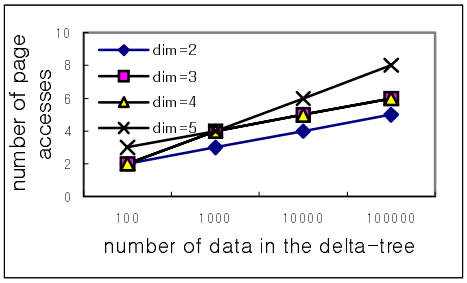


Figure 12. Insert performance

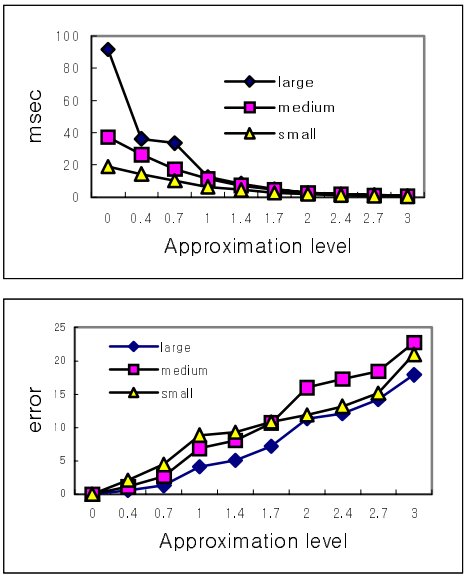


Figure 13. Performance and error rates for uniform distribution, dim=4, query sizes=large, medium, small, and number of data =10000

Figure 13 shows the error rate of approximate query results by the hybrid method. Experimental parameters are of the dimensionality 4, uniform data distribution, and 10000 updates stored in the Δ-tree. The X-axis represents the approximation level of the Δ-tree up to which the approximate query is performed. The level of data nodes (i.e. leaf nodes) is 0 and their parents have level 1, and so on. The approximation level 0 indicates that the query is evaluated up to the data node, and the approximation level 1 indicates that the query is evaluated up to the level 1. The approximation level 0.4 (=0.6×0+0.4×1) indicates that the query is evaluated up to the level 1 for 40% of nodes at the level 1, and up to the level 0 for 60% of nodes at the level 1. Similarly, the approximation level 1.7(= 0.3×1+0.7×2) indicates that the query is evaluated up to the level 2 for 70% of nodes at the level 2, and up to the level 1 for 30% of nodes at the level 2. For nodes whose differences of approximation values (see Section 4.2) are bigger, the search goes down to 1 lower level.

As shown in Figure 13, the evaluation time is reduced rapidly as the approximation level becomes higher, while the error rate increases slightly. Therefore, we can obtain the high performance within a reasonable error rate if we select an appropriate approximation level. Figure 14 shows the case that the Δ-Tree has 50000 data. Compared to Figure 13 which is the case that the Δ-Tree has 10000 data, the error rate is decreased as more data are stored in the Δ-Tree, while the query performances are decreased.

Figure 15 shows the experimental results with the uniformly distributed data. The number of data is 50000, the query size is large, and the dimensionality is varied from 2 to 5. Figure 16 shows the experimental results with the Zipf distributed data. The number of data is 50000 and the dimensionality is varied from 2 to 4. As shown in Figure 15 and 16, approximation levels between 1 and 2 are observed to be good points for the hybrid method.

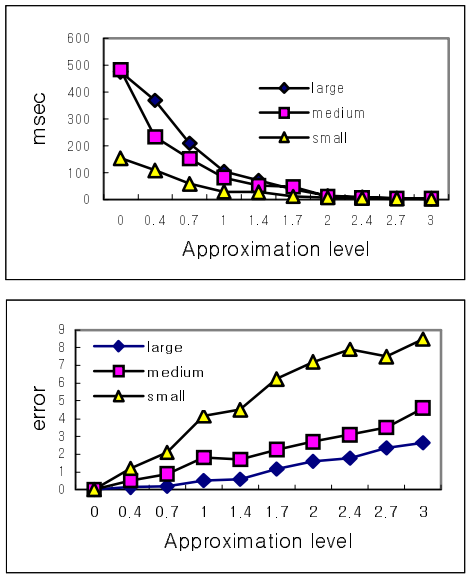
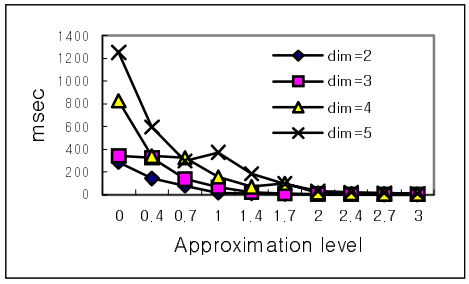


Figure 14. Performance and error rates for uniform distribution, dim=4, query sizes = large, medium, small, and number of data in the Δ-tree = 50000.



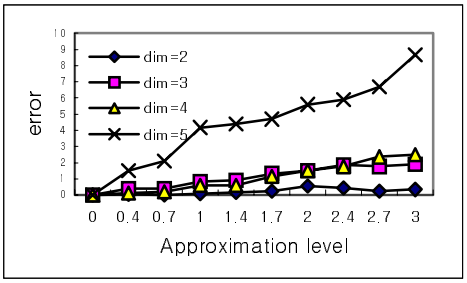


Figure 15. Performance and error rates for Uniform distribution, dim=2,3,4,5, query size = large, and number of data in the Δ-tree = 50000

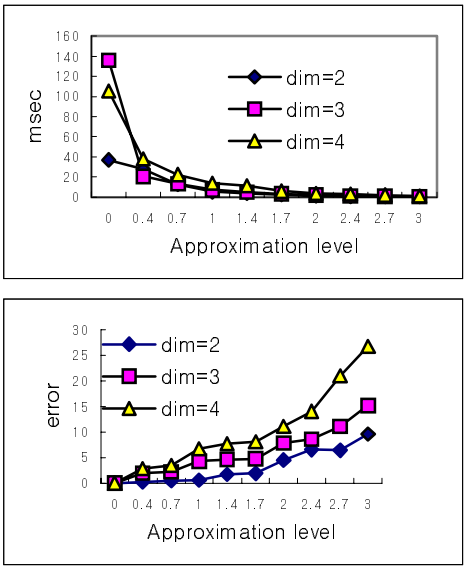


Figure 16. Performance and error rates for Zipf distribution, dim=2,3,4, query size = large, and number of data in the Δ-tree = 50000

6. Conclusion

In this paper, we proposed a new technique called a dynamic update cube which is designed to reduce the update cost of the data cube significantly, while maintaining reasonable search efficiency. In the recent dynamic enterprise environment where data elements in the data cube are frequently changed, the response time is affected by the update cost as well as the search cost of the cube. We exploited a hierarchical data structure, called the Δ-tree, to store the information of updated cells in the data cube and thus to minimize the update cost since only a small fraction of data elements in the data cube is changed. In addition, by taking advantages of the hierarchical tree structure of the dynamic update cube, we proposed a hybrid method to provide either an approximate result or a precise one to reduce the overall cost of queries. It is useful for diverse applications that need quick approximate answers rather than accurate ones, such as decision support systems.

The update complexity of our method is $O(\log N_u)$, with respect to N_u, the number of changed cells in the data cube. We have also provided experimental evaluations on the query and update efficiency and on the approximation accuracy and efficiency, with respect to various dimensions and query sizes. Experimental results demonstrate that our method performs very efficiently for update and query operations, and show reasonable approximation error rates with a significant gain in speed when the hybrid method is used.

As the future work, we plan to investigate techniques to further reduce the approximation error of the hybrid method, and to develop indexing mechanisms for high-dimensional (e.g, 10 and 20 dimensions) data cubes.

References

[BKK96] S. Berchtold, D. Keim, and H. Kriegel, The X-tree: an index structure for high dimensional data, Proceedings of Int'l Conference on Very Large Data Bases, India, 1996, pp. 28-39.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, Proceedings of ACM SIGMOD Int'l Conference on Management of Data, New Jersey, 1990, pp. 322-331.

[BS97] Alex Berson, Stephen J. Smith, Data WareHousing, Data Mining, & OLAP, McGrawHill, 1997.

[CI99] C.-Y. Chan, Y. E. Ioannidis, Hierarchical cubes for range-sum queries, Proceedings of Int'l Conference on Very Large Data Bases, Scotland, 1999, pp. 675-686.

[Cod93] E. F. Codd, Providing OLAP(on-line analytical processing) to user-analysts: An IT mandate, Technical report, E. F. Codd and Associates, 1993.

[GAE00] S. Geffner, D. Agrawal, A. El Abbadi, The Dynamic Data Cube, Proceedings of Int'l Conference on Extending Database Technology, Germany, 2000, pp.237-253.

[GAES99] S. Geffner, D. Agrawal, A. El Abbadi, T. Smith, Relative prefix sums: an efficient approach for quering dynamic OLAP Data Cubes, Proceedings of Int'l Conference on Data Engineering, Australia,1999, pp. 328-335.

[Gut84] A.Guttman, R-trees: a dynamic index structure for spatial searching, Proceedings of ACM SIGMOD Int'l Conference on Management of Data, Massachusetts, 1984, pp. 47-57.

[HAMS97] C. Ho, R. Agrawal, N. Megido, R. Srikant, Range queries in OLAP Data Cubes, Proceedings of ACM SIGMOD Int'l Conference on Management of Data, 1997, pp. 73-88.

[LWO00] W. Liang, H. Wang, M. E. Orlowska, Range Queries in dynamic OLAP data cubes, Data & Knowledge Engineering 34, 2000, pp. 21-38.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos, The R+-tree: a dynamic index for multi-dimensional objects, Proceedings of Int'l Conference on Very Large Data Bases, England, 1987, pp. 507-518.