Efficient Aggregation over Multidimensional Objects

Donghui Zhang* Vassilis Tsotras† Alexander Markowetz‡
Marios Hadjieleftheriou§ Dimitrios Gunopulos¶ Bernhard Seeger||

Abstract

Multi-dimensional aggregation queries are very useful but costly operations. Previous approaches on multi-dimensional aggregations works either for point objects (the Dynamic Data Cube, etc.), or for one-dimensional interval objects (the SB-tree, etc.). This paper proposes efficient techniques to compute and incrementally maintain multi-dimensional aggregates for objects which may have non-zero extent in any dimension(s). We propose a new index structure called the BA-tree which absorbs features from both the Dynamic Data Cube and the SB-tree. We further examine multi-dimensional aggregations where object values are not constant but are described by functions. This has a non-trivial consequence in aggregation computations. An object's participation in the aggregation is the function integral over the intersection between the object's multidimensional extent and the query. We show that the BA-tree paradigm can be extended to solve functional multi-dimensional aggregates for a broad class of functions. Finally we present experimental results which prove the efficiency of the proposed techniques.

1 Introduction

In this paper we introduce the *box aggregation problem*, which is concerned with the computation of orthogonal range-sum queries over objects with non-zero extents in n-dimensional space. Formally, it is stated as: "*given n weighted rectangular objects and a query rectangle r in the d-dimensional space, find the cumulative weight of all the objects which intersect r*". This is a far more general problem than the ones looked at before by other authors ([HAM+97, VWI98, CI99, GAE+99, VW99, SFB99, GAE00]). In particular, previous approaches consider only point objects (i.e., objects with zero extent in all dimensions). Moreover, these points had to fall on a fixed multidimensional grid. Recently, the work in [YW01, ZMT+01] addresses aggregations over interval objects but not in multiple dimensions. We will use the term 'box aggregation' since in recent literature, the term 'multi-dimensional' aggregation has been also used in the context of data cube and group-by computations. ([AAD+96, HHW97, RKR97, ZDN97, AGP00]).

The box aggregation problem has many real-life applications. For example, consider a database in an agricultural or environmental agency that keeps track of pesticide usage. Each record represents the treatment of an area over a certain time period. It contains a 3-dimensional rectangle (that is, a 2-dimensional area describing the orchard or field which is sprayed and the corresponding time interval) and a value (the volume of the pesticide). An example of a box aggregation query is: "*find the average volume of pesticide per orchard in Orange County for March 1999*".

A straightforward solution to the box aggregation problem is to index the records using some spatial access method (SAM) like the *R^*-tree* ([BKS+90]). The aggregation is computed incrementally as records in the query box are identified (through a range search). However, this approach is greatly affected by the size of the query box. In the worst case, all the objects in the database need to be examined to compute a single aggregation.

*Computer Science Department, University of California, Riverside, CA 92521. donghui@cs.ucr.edu
†Computer Science Department, University of California, Riverside, CA 92521. tsotras@cs.ucr.edu
‡Fachbereich Mathematik & Informatik, Philipps Universität Marburg, Germany. alexander@markowetz.de
§Computer Science Department, University of California, Riverside, CA 92521. marioh@cs.ucr.edu
¶Computer Science Department, University of California, Riverside, CA 92521. dg@cs.ucr.edu
||Fachbereich Mathematik & Informatik, Philipps Universität Marburg, Germany. seeger@Mathematik.Uni-Marburg.de

In this paper, we propose an efficient solution to the general box aggregation problem. Our method is based on a new, disk-based and dynamic index structure called the *Box Aggregation Tree (BA-tree)*. We note that our solution applies also to computing range-sums over datacubes. The best known solution for data cube range-sum is [GAE00]. When applied to this problem, the BA-tree differs from [GAE00] in two ways. First, it is disk-based, while [GAE00] presents an in-memory structure. Second, the BA-tree partitions the space based on the data distribution while [GAE00] does partitioning based on a uniform grid.

Furthermore, this paper considers a variation called the *functional box aggregation* problem, where object values are not constant but are described by functions. This implies that an object's participation in the aggregation result is the function integral over the intersection between the object's box and the query box. This approach allows different kinds of sum-aggregates, for which no efficient aggregate index has been proposed. This functional aggregate has great potential for practical applications.

As an example, consider the above pesticide spraying case. In the box aggregation problem, the volume associated with each spray is constant. This means that for any given query, the value either contributes to the query result as a whole or not at all, regardless of how large the intersection between the record and the query box is. The agency may need to keep the volume of each record as a function of litres per square acre per day, and to answer queries of the form: *"find the total volume of pesticides sprayed in Orange County in March 1999"*. Each tuple will only contribute to the query in as much, as its associated area overlaps with the query rectangle. The latter query is an example of the functional aggregation problem.

We show that for a special set of functions, we can extend the BA-tree to solve the functional aggregate query. To the best of our knowledge, this is the first efficient solution to this problem.

The main contributions of this paper are:

- We propose a new, dynamic, disk-based index to compute orthogonal range-sum queries over multidimensional objects with non-zero extents.
- We propose an interesting and novel variation, i.e., functional aggregates, and present a solution for a general class of functions.
- As by-products of this work we present two optimization techniques for using a plain R-tree for the box aggregation query and the functional aggregation respectively.

The rest of the paper is organized as follows. Section 2 discusses related previous work. In section 3, the box aggregation problem is reduced to the simpler problem of designing an index with specific requirements. Section 4 presents the BA-tree which meets these criteria; moreover, this section discusses the aggregate-R-tree alternative. Section 5 presents our solution to the functional box aggregation problem. Results from our experimental comparisons appear in section 6. Finally, section 7 provides conclusions.

2 Related Work

The orthogonal point range-sum problem has received vast attention in the past 20 years in the field of computational geometry [Meh84, PS85, Mat94, AE98]. Most of the earlier solutions utilize some variation of the range-tree ([Ben80]) following the multi-dimensional divide-and-conquer technique. The best main-memory solution is a compressed range-tree proposed in [Cha88].

The data cube range-sum problem (i.e., data points are on a grid) has been addressed in [HAM+97]. It is proposed to solve the problem by maintaining an array of *prefix-sums*. Assume that the original d-dimensional data cube array A is bounded by two points $p_{low} = (0, \cdots, 0)$ and $p_{high} = (n_1 -$

$1, \cdots, n_d - 1)$, where n_i characterizes the number of different locations A has in dimension i. A prefix-sum array P of the same size, is maintained where

$$P[l_1, \cdots, l_d] = \sum_{i_1=0}^{l_1} \cdots \sum_{i_d=0}^{l_d} A[i_1, \cdots, i_d];$$

The range-sum query can then be transformed into 2^d array look-ups in P and their result can be combined (through additions and substractions) as illustrated in figure 1.

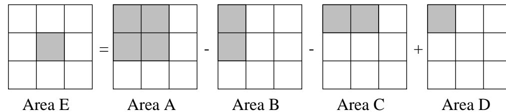


Figure 1: A range-sum query in array A can be transformed into the combination of 2^d values in P.

However, this solution has very expensive update. In the worst case, if $A[p_{low}]$ is changed, the whole array P needs to be modified at $O(n)$ update cost. In an effort to reduce the update penalty, [GAE+99] proposes to keep not the prefix-sums, but 'relative-sums'. This reduces the update cost to $O(\sqrt{n})$. A relative-sum is the prefix sum of a cell relatively to some other cell, not necessarily the lowest cell in the array. [CI99] improves the work of [GAE+99] by storing the cube in a hierarchy.

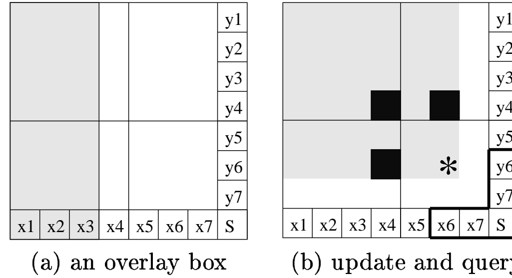
The best solution to this problem so far is the *dynamic data cube* ([GAE00]), which reduces the update to $O(\log^d n)$ by increasing the query time from $O(1)$ (assuming d is constant and each cell of the array A can be accessed at constant time) to $O(\log^d n)$. Since our method can be seen as an extension of the dynamic data cube we present a summary of this structure below.

The idea behind the dynamic data cube is to form a recursive orthogonal partitioning of array A into sub-arrays called *overlay boxes*. The overlay boxes are then organized into a tree structure. Assuming that A has the same number m of rows in all dimensions, the tree has $\log_2 m$ levels. At the highest level, the tree has one root, which corresponds to the whole region of A. At the leaf level, each node corresponds to a cell from A. In each overlay box, as shown in figure 2a, the structure stores d *row-sum* value lists (the x's and y's in the figure) and a *subtotal* value. Each row-sum value stores the sum of all cells of A which are from the smallest cell of this overlay box to the cell containing the row-sum. E.g. row-sum x_3 contains the sum of all cells in the shaded area. The *subtotal* contains the sum of all cells in the overlay box.

A range-sum query is transformed into 2^d prefix-sum queries, where each prefix-sum query computes the prefix-sum at some cell (or point) p. To answer a prefix-sum query regarding point p, the dynamic data cube will follow a single path from the root to the leaf that corresponds to p. At the root level, the prefix sum is also the relative-sum of p with regard to the lowest array cell. This relative-sum can be decomposed into several relative-sums, one for each child region. Consider for example computing the relative-sum for the cell p marked with a star in figure 2b. Since p falls into one and only one child region, the relative-sum for p in this region can be computed recursively by querying this child node. The relative-sum for the other child regions can be acquired directly from their subtotals or their row sum values, as illustrated by the three dark cells in figure 2b.

Similarly, an update affects a single path from root to leaf. It first finds the leaf node corresponding to the cell which is to be updated, then updates the row-sums and subtotals of ancestor nodes of the leaf node in a bottom-up fashion. Assume an update is to be applied on the same cell marked by a star in figure 2b. The update affects the row-sums values (in this example, cells x_6, x_7, y_6, y_7) whose index is no smaller than that of the update point.

Related is also the work in [VWI98, VW99, SFB99] which instead of the whole data cube store an approximation and thus give only approximate answers to range-sum queries.



(a) an overlay box (b) update and query

Figure 2: Illustration of the dynamic data cube.

All the above work on the range-sum computation focuses on point data. Aggregations over objects with no-zero extents have been examined in [YW01, ZMT+01]. Recently, [YW01] developed an efficient solution to compute aggregates over 1-dimensional interval objects. A new, disk-based structure called the *SB-tree* is proposed. Since our approach can be considered as a generalization of the SB-tree, we summarize its characteristics below. The SB-tree incorporates properties from both the segment tree ([PS85]) and the B-tree. The segment tree features ensure that the index can be updated efficiently when tuples with long intervals are inserted or deleted. The B-tree properties make the structure balanced and disk-based. Conceptually the SB-tree indexes the time domain of the aggregated tuples. Each interior tree node contains between $b/2$ and b records, each record representing one contiguous time interval. For each interval, a special value is also kept in the record that will be used to compute the aggregate over this interval. Intervals are kept in both interior and leaf nodes. Moreover, the overall interval associated with a node contains all intervals in the node's subtrees.

An advantage of [YW01] is that an instantaneous temporal aggregate is computed by recursively searching the SB-tree (starting from the root) and accumulating the aggregate value along the tree nodes visited. This results in fast aggregate computation time, namely, $O(\log_b n)$. Note that a special "compaction" algorithm is also presented that merges leaf intervals with equal aggregate values. This can reduce the height of the tree and hence its aggregate computation to $O(\log_b m)$.

The second advantage of the SB-tree is its fast update time, which is also logarithmic. The insertion of a new tuple with interval i and attribute value v is first directed into the root node. Each root record whose time interval is fully contained in i is updated by value v (the kind of update depends on the aggregate maintained by the SB-tree). Whenever interval i is partially contained by a root record, it is recursively inserted in the subtree under this root record. The SB-tree allows physically deleting tuples from the warehouse. Such a deletion is represented as an insertion of a new tuple with a negative attribute value v.

To support cumulative SUM, COUNT and AVG aggregates with arbitrary window offset w, two SB-trees are used, one maintaining the aggregates of records valid at any given time, while the other maintaining the aggregates of records valid strict before any given time. To compute the aggregation query, the approach first computes the aggregate value at the end of interval w. It then adds the aggregate value of all records with intervals strictly before the end of w and finally subtracts the aggregate value of all records with intervals strictly before the beginning of w. Finally, we note that a special extension of the SB-tree (the min/max SB-tree) can be used to support MIN and MAX aggregates, too.

In our previous work ([ZMT+01]), we have extended the work in [YW01] to efficiently maintain and compute the range-sum aggregates in the 2-dimensional space. We assume the transaction-time model. Each record has a key and a time interval and corresponds to two updates: an insertion at the start time of the interval and a logical deletion at the end time of the interval. The update (insertion/deletion) operations should happen at non-decreasing time order. The idea of our work is to follow the technique in [BGO+96] to make an SB-tree *partially persistent*.

None of the previous work efficiently solves the multi-dimensional range-sum aggregation problem in general. Also, we know of no previous work that would be comparable to the functional box aggregation.

3 Problem Reduction

We first reduce the range-sum query over multi-dimensional objects into a collection of point queries. Then we show that each of these point queries can be answered by an index that satisfies some criteria. By doing so, the range-sum problem is reduced to designing this index.

We differentiate between two types of dimensions.

Definition 1 *Consider a multi-dimensional space where each object is described by a box and a value. Dimension dim_i is called a* **point dimension** *if the projection of every object on dimension dim_i is a point; otherwise dim_i is called an* **interval dimension**.

Without loss of generality we assume that among the d dimensions there are d' interal dimensions and that the interval dimensions are before the point dimensions. Thus we have:

$$\underbrace{dim_1, \cdots, \mathrm{dim}_{d'}}_{\text{interval dims}}, \underbrace{dim_{d'+1}, \cdots, \mathrm{dim}_d}_{\text{point dims}}.$$

We define two one-dimensional queries, the L-query and the S-query. The L-query is defined for both point and interval dimensions; while the S-query is defined for the point dimension only.

Definition 2 *For a point dimension dim_v, an* **L-query** *regarding location l_v finds the sum of values of every record r, where the projection of $r.box$ to dim_v is smaller than l_v.*

Definition 3 *For an interval dimension dim_u, an* **L-query** *regarding location l_u finds the sum of values of every record r where the entirety of the projection of $r.box$ to dim_u is smaller than l_u; an* **S-query** *regarding location l_u finds the sum of values of every record r, where the projection of $r.box$ to dim_u contains l_u.*

For example, in a typical temporal database scenario, a record contains a key and a time interval. The time dimension is an interval dimension while the key dimension is a point dimension (figure 3). The L-query $L_{time}(t_2)$ is equal to 8, since the records whose interval is entirely before t_2 have values 1, 3 and 4. Similarly, $S_{time}(t_2) = 13$ and $L_{key}(k_2) = 18$.

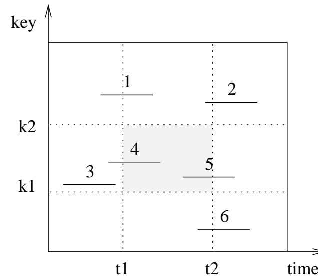


Figure 3: Illustration of various queries in the two-dimensional space.

By combining the S-queries and L queries in all dimensions, we can define a new kind of point query as follows.

Definition 4 *Given an instance of the following class of queries $(L|S)_1 \cdots (L|S)_{d'} L_{d'+1} \cdots L_d$ and a query point $p = (l_1, \cdots, l_d)$, the* **point aggregation query** *regarding p returns the sum of every record r where the projection of $r.box$ to every dimension k satisfies the $(L|S)_k$-query condition regarding l_k.*

For example, in figure 3, $SL(t_2,\ k_2) = 7$ and $LL(t_2,\ k_2)$=11.

Theorem 1 shows that a range-sum query is transformed into point aggregation queries.

Theorem 1 *A range-sum query is transformed into* $3^{d'}2^{d-d'}$ *point aggregation queries.*

Proof. Assume each interval i is specified by two points $i.low$ and $i.high$. We first show that theorem 1 is correct for the one-dimensional case (d=1). If the dimension is an interval dimension, the range-sum regarding query interval i is equal to the sum of all records whose interval contains $i.high$ plus the sum of all records whose entire interval is before $i.high$, minus the sum of all records whose entire interval is before $i.low$. More formally,

$$SUM(i) = S(i.high) + L(i.high) - L(i.low) \quad \text{for interval dim;} \tag{1}$$

If the dimension is a point dimension, the range-sum regarding query interval i is equal to the sum of all records before $i.high$ minus the sum of all records before $i.low$. More formally,

$$SUM(i) = L(i.high) - L(i.low) \quad \text{for point dim;} \tag{2}$$

In both cases, the theorem holds.

For higher dimensions, we decompose the range-sum query one dimension at a time. To decompose the range-sum query for a certain dimension dim_k, we assume the records satisfying the range condition for the rest of the dimensions can be found and we decompose the query on dim_k using equations 1 and 2. Each decomposition on an interval dimension multiplies the number of queries by 3 and each decomposition of a point dimension multiplies the number of queries by 2. Thus after all the d dimensions are decomposed, we get $3^{d'}2^{d-d'}$ point aggregation queries. □

As an example, consider the scenario shown in figure 3. Our goal is to compute the range-sum of the range from $(t_1,\ k_1)$ to $(t_2,\ k_2)$ which is the shaded area. First, we decompose the time dimension. Among all the records whose keys are between k_1 and k_2, equation 1 holds. Or,

$$SUM((t_1,\ k_1),(t_2,\ k_2)) = S_{(k_1,k_2)}(t_2) + L_{(k_1,k_2)}(t_2) - L_{(k_1,k_2)}(t_1)$$

For each of the three new queries, we decompose on the key dimension. For example, to compute $S_{(k_1,k_2)}(t_2)$ which is the sum of records whose intervals contain t_2 and whose keys are between k_1 and k_2, we can focus on all records whose intervals contains t_2 and use equation 2. Thus we have

$$S_{(k_1,k_2)}(t_2) = SL(t_2,\ k_2) - SL(t_2,\ k_1)$$

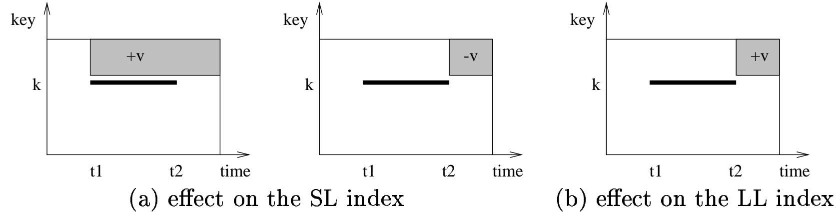
After we decompose the other two queries similarly, we get:

$$\begin{aligned} SUM((t_1,\ k_1),\ (t_2,\ k_2)) &= SL(t_2,\ k_2) + LL(t_2,\ k_2) + LL(t_1,\ k_1) \\ &- SL(t_2,\ k_1) - LL(t_2,\ k_1) - LL(t_1,\ k_2) \end{aligned}$$

The right side evaluates to $11 + 7 + 0 - 6 - 0 - 3 = 9$, which is the correct answer.

Now let's consider what properties an index structure should have so as to compute point aggregation queries. From definition 4, it is easily seen that we have $2^{d'}$ different forms of point aggregation queries. Thus we maintain an index for each of them. We claim that all of these indices can be of the same kind. Let's consider the two-dimensional example again. We need to maintain an index for the SL query (the SL index) and an index for the LL query (the LL index). Suppose a new object is introduced whose rectangle is from $(k,\ t_1)$ to $(k,\ t_2)$ and whose value is v. The effect of this new object on the SL index and the LL index is shown in figure 4a and 4b, respectively.

Hence the problem to efficiently maintain and compute range-sums over multi-dimensional objects is reduced to the problem of designing an efficient index structure which:



(a) effect on the SL index (b) effect on the LL index

Figure 4: Transform an object insertion into insertions into the point aggregation indices.

- logically stores a value at each point p;
- supports the following insertion operation (*BA insertion*): given point p and value v, add v to the values at all the points in the region from p to p_{max}; and
- supports the following point query (*BA point query*): given point p, find the value at p.

In the one-dimensional case, the SB-tree fulfills these requirements. We present an index structure in higher dimensions which satisfies the requirements in the next section.

4 The BA-tree

In this section we present a new index structure, the *Box Aggregation Tree (BA-tree)*, which efficiently supports the BA insertion operation and the BA point query.

The properties of a d-dimensional BA-tree are summerized below:

- Every index or leaf record is associated with a box. The boxes of all records in a page form a non-intersecting full partitioning of the box for this page.
- Every leaf record is of the form: $\langle box, value \rangle$.
- Every index record contains a box, a pointer to a child page and some additional information which will be explained later in this section.

To get a basic understanding of the BA-tree, let's first consider a naïve balanced tree index which has the same properties as those of the above BA-tree without the additional information in the index records. This naïve tree supports both the BA point query and the BA insertion. To answer a BA point query regarding point p, we start from the root node and follow the child pointer whose box contains p until we find the leaf record whose box contains p; the value of this leaf record is the query result. To perform a BA insertion regarding point p and value v, the tree is browsed in a top-down fashion. For each node accessed, all records in it whose box intersects with the box (p, p_{max}) are examined. Eventually for every intersecting leaf record, if the box of the record is contained in (p, p_{max}), the value of it is increased by v. Otherwise the record is split into several records. One of the resulting records will have a box equal to the intersection between the original record box and (p, p_{max}). The value of this record is increased by v. The problems with this approach are: (1) The insertion is expensive, since in the worst case all the index records need to be examined and all the leaf records need to be updated. (2) It occupies too much space; trivial analysis shows that the space complexity is $O(n^2)$ where n is the number of insertions taken place.

We now show how the BA-tree can utilize some additional information to dramatically improve the performance of the above naïve tree. First, each record contains a *subtotal* value. We also want to maintain values for the borders of a record's box. In particular, we are interested on the lower borders of the record's box. Each of the $d/2$ lower borders is initialized as a single segment with value 0. As insertions are processed, a lower border can be segmented into a set of smaller segments, each

with a different value. We use the variable $lowborder_i$ to represent the values maintained over the lower border on the i-th dimension of an index record. To summarize, an index record in a BA-tree is of the form: $\langle box, child, subtotal, lowborder_1, \cdots, lowborder_d \rangle$.

Figure 5 shows an example of a BA-tree index page which contains five records. Suppose an insertion with value v takes place at point p which is represented as shown by a black dot in the figure. The BA insertion operation requires adding value v to all points in the shadowed box. Both the naïve tree and the BA-tree examines the records A, C, D and E whose boxes intersect the shadowed box. The naïve approach will recursively update the subtrees rooted by each of these records. The BA-tree, however, recursively updates only one of them, the subtree (in fact, only a path in the subtree) under record C (whose box contains p). For the other intersecting index records, the boxes of them are either fully contained or partly contained in the box (p, p_{max}). For a fully contained one (e.g. record D), we add the value v to its subtotal field. For a partly contained one (e.g. record A and E), we add v to one of its lower borders as shown by the thick line segments in the figure. So in a BA-tree each insertion modifies only one path from root to leaf. Morever, if the insertion point p falls onto the lowborder of some index record being examined, the insertion algorithm will stop at that level; there is no need to examine any subtree.

The BA point query algorithm should be modified from the naïve index accordingly. A point query in a BA-tree visits a single path from root to leaf, following at each level the child pointer whose box contains the query point. For each index record it examines, the subtotal and the low-border value regarding the query point in every dimension is added to the result. When the algorithm reaches the leaf record whose box contains the query point, the value saved at the leaf record is added to the result. As an example, suppose the query point p falls into the index page as shown by a star in figure 5. The previous insertion should affect the value at p. However, the insertion did not modify the leaf record whose box contains p. Instead, it modifed a low border of E. The query result as produced by the above point query algorithm is correct because this low border is examined and the corresponding value is added to the query result.

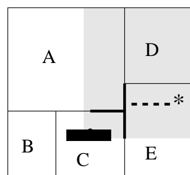


Figure 5: Illustration of an index page of a BA-tree.

It remains to show how a low border associated with some index record can be maintained and queried efficiently. If the low border is small, we can simply store it inside the page containing the index record. If it gets too large, especially in the higher-dimensional spaces, we need to store it as a secondary-memory structure. In the latter case, an interesting observation is that a d-dimensional low border is a $(d-1)$-dimensional index structure which should support the BA point query and the BA insertion operation efficiently. So in general, a low border associated with a d-dimensional index record can be implemented as a $(d-1)$-dimensional BA-tree. E.g. in figure 5, we store a low border inside an index page as an array of non-intersecting intervals, each of which has a value. However, if any low border becomes too big, we can store it as a 1-dimensional BA-tree, which is an SB-tree ([YW01]).

More formally, we give the insertion and point query algorithms of the BA-tree as follows.

Algorithm *Insert*(Point p, Value v)

1. N = root page;
2. while N is not leaf do
3. $\quad r_0$ = the record in N whose box contains p;

```
4.      for every record r in N
5.          if r.box is contained in the box (p : p_max) then
6.              r.subtotal += v;
7.          else if r.box intersects with (p : p_max) and r ≠ r_0 then
8.              add v to the corresponding lowborder of r;
9.          endif
10.     endfor
11.     if p is on some border of r_0 then
12.         add v to the subtotal or the corresponding lowborder of r_0;
13.         GoUp( N );
14.         return;
15.     else
16.         N = ReadPage(r_0);
17.     endif
18. endwhile
19. for every record r in N
20.     if r.box is contained in (p : p_max) then
21.         r.value += v;
22.     else if r.box intersects with (p : p_max) then
23.         split r into several copies and add v to the corresponding copy;
24.     endif
25. endfor
26. Merge records in N if possible;
27. GoUp( N );
```

Algorithm *GoUp*(Page N)

```
1. while N is not root
2.     if N overflows then
3.         Split N into two and insert the new copy into the parent;
4.     endif
5.     set N = the parent of N;
6. endwhile
7. if N overflows then
8.     create new root;
9.     Split N into two and insert references to the new root;
10. endif
```

Algorithm *PointQuery*(Point p)

```
1. N = root page;
2. while N is not leaf
3.     r = the record in N whose box contains p;
4.     v += r.subtotal;
5.     for every lowborder lb of r
6.         v += the border value of p regarding lb;
7.     endfor
8.     if p is on some border of r then
9.         return v;
10.    else
11.        N = ReadPage(r);
12.    endif
13. endwhile
14. r = the record in N whose box contains p;
15. v += r.value;
16. return v;
```

4.1 Discussion

To incrementally maintain an access method for the d-dimensional box aggregation queries, our proposed technique maintains $2^{d'}$ BA-trees, where d' is the number of interval dimensions. Note that in many applications $d' \ll d$. Our approach applies to the computation of range-sums in OLAP arrays, where $d' = 0$, in which case only a single BA-tree is needed. Each range-sum query is then transformed to $3^{d'} 2^{d-d'}$ point queries in the BA-trees. Since a BA-tree is unbalanced, the worst case update and query performance is linear. However, in practice the BA-tree behaves as a balanced structure. Since each update and query visits a single path of the tree, its average update and query performance are both logarithmic.

A straightforward alternative is to index all the objects in a multi-dimensional index like the R*-tree. The problem is that for a large query rectangle, many objects need to be examined. We hereby propose an optimization for the straightforward approach as follows. We store a value with every index record r of a R*-tree as the sum of values of all the objects stored in the subtree rooted by the node which is pointed to by r. Whenever we insert an object into a subtree, the value associated with the index record pointing to the subtree is updated. This optimization improves the range-sum query in that if the query box contains the box of an index record, the whole sub-tree does not need to be examined since the sum of all records in the sub-tree is available.

5 The Functional BA-tree

The functional BA-tree provides a new kind of aggregation query. Instead of storing fixed values in each entry, it stores functions.

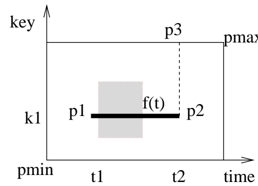


Figure 6: Illustration of the functional box aggregation problem.

Figure 6 shows a 2-dimensional scenario where the time is represented as an interval dimension and the keys as a point dimension. We see an object, whose box is bound by $p_1 = (t_1,\ k_1)$ and $p_2 = (t_2,\ k_1)$. The shadowed area resembles a range-sum query box. In an ordinary BA-tree, an object would either fully contribute to a value, or not at all. The former would be the case in this example. What, if instead of having this object fully contribute to the query, we wanted it to contribute only to the same degree to which it overlaps the query rectangle? Instead of a fixed value, we assign a function to each object, and evaluate the integral of this funtion over the area, where it overlaps the query rectangle. Since it is the integrals we are interested in, we store the integral of the function $f(t)$ with each tuple and not the function itself.

In contrast to the case in section 4, we can now regard the insertion of a record as the independent insertions of (infinitely) many points, i.e. for all the points in the box of the record. Thus, the query answer methodology for point data as shown in figure 1 can be applied. Again, the methodology means that a range-sum query is transformed into 2^d prefix-sum queries, where each prefix-sum query regarding point p means to compute the range-sum of the region from p_{min} to p. To efficiently support the prefix-sum queries, we plan to maintain an index which logically stores at every point p a value as the prefix-sum at p. We now consider how we should modify the index when a new object is inserted. In the 2-dimensional scenario as shown in figure 6, the effect of inserting a new record with box being $(p_1,\ p_2)$ and with value function being $f(t)$ is twofold:

- for every point $(t,\ k)$ in the region $(p_1,\ p_3)$, the value $\int_{t_1}^{t} f(t')dt'$ needs to be added to the point;

- for every point $(t,\ k)$ in the region $p_2,\ p_{max}$, the value $\int_{t_1}^{t_2} f(t')dt'$ needs to be added to the point.

These two insertions are equivalent to the following two insertions:

- for every point $(t,\ k)$ in the region $(p_1,\ p_{\max})$, add the value $\int_{t_1}^{t} f(t')dt'$;
- for every point $(t,\ k)$ in the region $(p_2,\ p_{max})$, add the value $\int_{t_1}^{t_2} f(t')dt' - \int_{t_1}^{t} f(t')dt'$.

In general, in the d-dimensional space where d' dimensions are interval dimensions, the function associated with each object is a d'-variable function $f(x_1, \cdots x_{d'})$. To maintain an index which logically stores the prefix-sum of every point in the d-dimensional space, for each object insertion, we need to perform $2^{d'}$ insertions on the index. Each of the insertions adds the value of some function $h(x_1, \cdots x_{d'})$ to all points in some region whose low point is one of the object box's corner points and whose high point is p_{max}. Assume the projection of the low point and high point of the object box to an interval dimention k is $p_1[k]$ and $p_2[k]$. Respectively, the function $h(x_1, \cdots x_{d'})$ is the $\pm$ aggregation of some function $hterm$:

$$hterm(x_1, \cdots x_{d'}) = \int_{p_1[1]}^{v_1} \cdots \int_{p_1[d']}^{v_{d'}} f(x'_1, \cdots, x'_{d'})dx'_{d'} \cdots dx'_1,$$

where v_k is either the constant $p_2[k]$ or the variable x_k.
We note that all the $2^{d'}$ insertions have the form of the BA insertion since the high point of the box is always p_{max}. The only difference is that instead of adding constant values, we now add functions. We can therefore use a BA-tree in this environment and thus solve the functional box aggregation problem. However, the function $hterm(x_1, \cdots, x_{d'})$ needs to satisfy the following requirements:

- it can easily aggregated using $+$ or $-$ operators;
- any thus aggregated function is *compact*, i.e. can be represented in constant space, that is the set of functions is closed under the + and - operator;
- given a point, the function value can be easily computed;

It is hard to determine all the sets of functions that fullfill these requirements. We will however provide the most important example:

$$F' = \{f_1(x_1) \cdots f_{d'}(x_{d'}) \mid f_k(x) \text{ is a polynomial of rank at most } r_k \text{ for } k \in [1,\ d']\}$$

If $f \in F'$, $hterm$ can be decomposed as follows. Let $g_k(x) = \int f_k(x)dx$. We have

$$hterm(x_1, \cdots x_{d'}) = (\int_{p_1[1]}^{v_1} f_1(x')dx') \cdots (\int_{p_1[d']}^{v_{d'}} f_{d'}(x')dx') = \prod_{k=1}^{d'} (g(v_k) - g(p_1[k])).$$

Since $g_k(x)$ is a polynomial with rank at most $r_k + 1$, the above $hterm$ can obviously be stored in constant space by storing only the coefficients. The $+$ or $-$ aggregations of multiple $hterm$s can be done easily by performing aggregations on the coefficients and the resulting term uses the same amount of space for storage, thus the set of functions is closed.
To sum up, by storing and aggregating functions instead of constant values in the BA-tree, we have solved the functional box aggregation problem. In practice, this query might prove to be quite interesting. In the hospital-example they used, [YW01] showed in the two dimensional case, how to compute the sum over all prescriptions that were alive during some period during the query interval. However, is might be more interesting to answer the query "how many pills were actually handed out during a time period"? A functional BA-tree, or functional SB-tree for simplicity, will be able to resolve such queries. Eventually, many other usefull applications, especially with polynoms of higher degrees will arise.

5.1 Discussion

Unlike the multi-dim aggregation problem without functions, to incrementally maintain an access method for the functional problem we need only a single Functional BA-tree.

A straightforward approach is still to index all the objects using an R*-tree. An aggregation query is then reduced to a selection query, where for each record that satisfies the selection condition, the value to be contributed by this record is computed and added to the result on-the-fly. We propose an optimization to this straighforward approach. The idea is, we pre-determine a grid in the multi-dimensional space. For example, in the one-dimesional case, we pre-determine a grid of time instants. We associate a value for each of the grid point p as the sum of values of all points in the box from p_{min} to p. The grid should be so small that it can be kept in memory. To insert/delete a new record, besides updating the R*-tree, we also update the grid values. To answer a range-sum query, if both the low point and the high point of the query box fall on the grid, the result is produced immediately by looking up the in-memory grid. Otherwise, we determine the two grid points which are the closest to the low and high points of the query rectangle and compute the range-sum over the box bounded by the two grid points as an approximate answer. This answer can be reported to the analyzer immediately. The approximate answer can then be refined gradually to perfection, by performing some range-sum queries on the R*-tree with smaller query boxes as the differences between the query box and the box bounded by the two grid points.

6 Performance

6.1 Experimental Setup

The algorithms are implemented in C++ using GNU compilers. The programs run on a Sun Enterprise 250 Server machine with two 300MHz UltraSPARC-II processors using Solaris 2.6. To compare the performance of the various algorithms we report the number of disk IOs.

Table 1 shows the algorithms we implemented.

Box Aggregation		Functional Box Aggregation	
R^*	Plain R^* tree	R^*	Plain R^* tree
R^*ba	optimized R^* tree	R^*fba	optimized R^* tree
2BAT	using two BA-trees	FBAT	the Functional BA-tree

Table 1: Implemented Algorithms

For the box aggregation problem, we compare the performance of the proposed BA-tree approach with the R*-tree based approaches. For brevity we report results from the two-dimensional case where one dimension is the interval dimension. We implemented the plain R*-tree which index all the records and use a range query to answer the aggregation query (denoted as R^*). We also compare the performance against the R*-tree approach which contains the optimization as proposed in section 4.1 (denoted as R^*ba).

For the functional box aggregation problem, we compare the performance of the proposed Functional BA-tree approach with the plain R*-tree based approach (denoted as R^*) and the optimized R*-tree approach, where the optimization is as discussed in section 5.1.

The buffering schemes we used in the implementation of the algorithms are as follows. For the BA-tree and the Functional BA-tree approach, we use a LRU buffer for each tree index. For the R*-tree approach, besides using a LRU buffer, for each tree we also buffer all the nodes along the path from the root to the most recently accessed node.

The datasets we use are created using the TimeIT software ([KS98]). Each dataset has 6 million objects, each of which has a two-dimensional line segment and a value. The space in both dimensions is [1, 1 million]. Each query reports the total number of IOs over 1000 randomly generated query

rectangles with fixed rectangle shape and size. The shape of a query rectangle is described by the *R/I ratio*, where R is the length of the vertical query interval divided by the length of the vertical space and I is the length of the horizontal query interval divided by the length of the horizontal space. The *query rectangle size (QRS)* is described by the percentage of the query area in the whole space. Unless otherwise stated, the default parameters we use are shown in table 2.

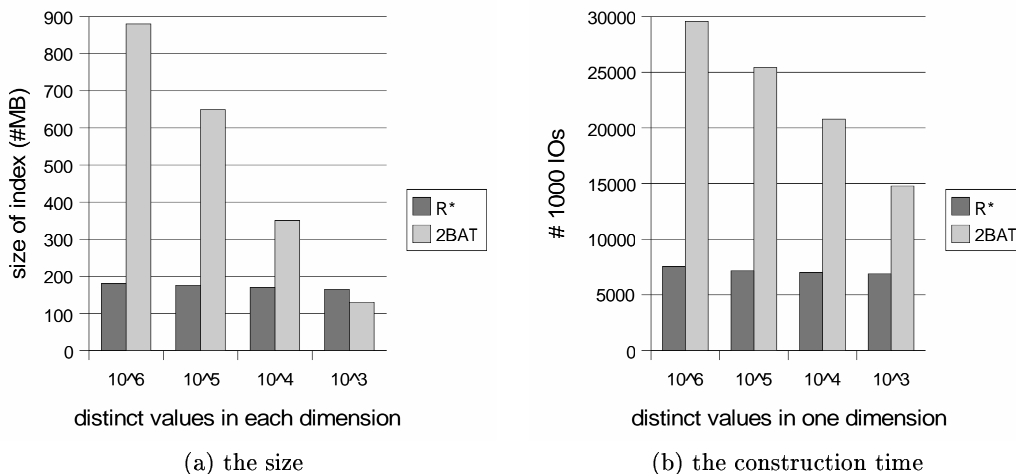
parameter	default value	parameter	default value
page size	4KB	memory buffer size	100 pages
R/I ratio	1 (square query)	QRS	1%

Table 2: Default parameters

6.2 Comparion of the size of the indexes and the update cost

Figure 7a shows the comparison of the size between the R*-tree and the combined size of the two BA-trees used for the Box Aggregation problem. The horizontal axis is the number of distict points each of the two dimension can have. Since each dimension has a value space from 1 to 1 million, each dimension can have 10^6 distinct integer valuess. We also generated datasets which assume each dimension has 10^5, 10^4 and 10^3 distinct points. We note that although the space requirement of the two BA-trees is generally more than that of the R*-tree, the space of the BA-tree drops rapidly as the distinct points per dimension drops. The reason is that if a point is inserted into the BA-tree and falls right onto the border of some index record, there's no need for the insertion process to recursively insert into the subtree pointed to by that record. When the number of distinct points drops to 1000, the combined space of the two BA-trees is less than the R* tree.

Figure 7b compares the index construction time. The BA-tree takes longer to build (about 2 to 4 times slower than the R*-tree).



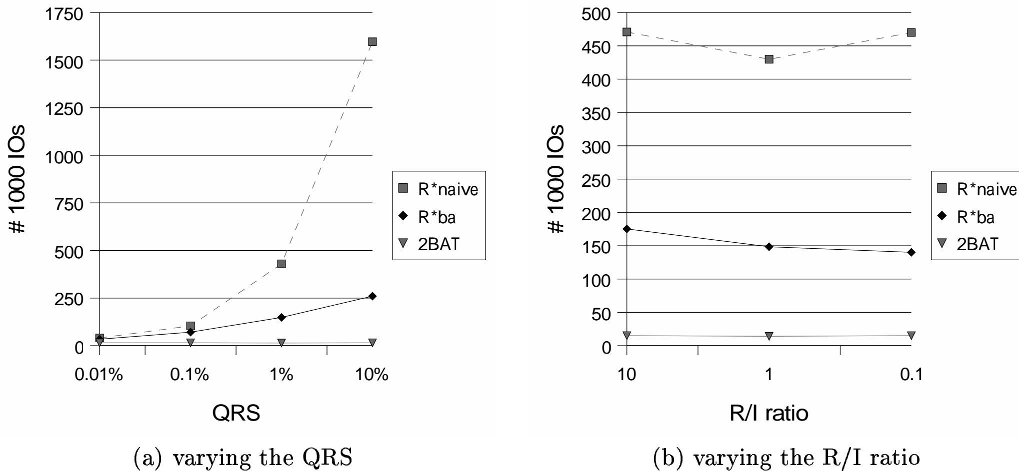
(a) the size (b) the construction time

Figure 7: Comparison of the occupied space and the construction time.

6.3 Performance of the Box Aggregation

Figure 8a shows the performance between a naïve R*-tree, an optimized R*-tree and the BA-tree approach. The BA-tree approach is clearly the best. This is to be expected, since on average we expect the BA-tree to have logarithmic query performance, while the R*-tree have linear query complexity. The optimized R*-tree is more advantegeous over the naïve one, especially if the query rectangle is large. Since a large query rectangle covers the rectangles of lots of internal nodes, the optimized R*-tree benefits by simply taking the value of the internal node and omit examining the sub-tree rooted by the node.

Figure 8b shows the performance of QRS=as the R/I ratio varies. Again, the optimized R*-tree approach is better than the naïve one but both are inferior the the BA-tree approach.



(a) varying the QRS (b) varying the R/I ratio

Figure 8: Comparison of the box aggregation performance.

6.4 Performance of the Functional Box Aggregation

Figure 9 shows the performance of various functional box aggregation algorithms. The Functional SB-tree is again the fastest. Note the there may be cases where the optimized approach is not as efficient as the naïve approach. This is because the optimized approach uses 2^d range-sum queries of smaller query rectangles to refine the approximate query result as given immediately by looking up the in-memory grid. Since these 2^d range-sum queries may need to examine the same records multiple times. Right now we maintain a 100*100 grid. Increasing the density of the grid will help in answering queries, but the CPU cost increases due to updates.

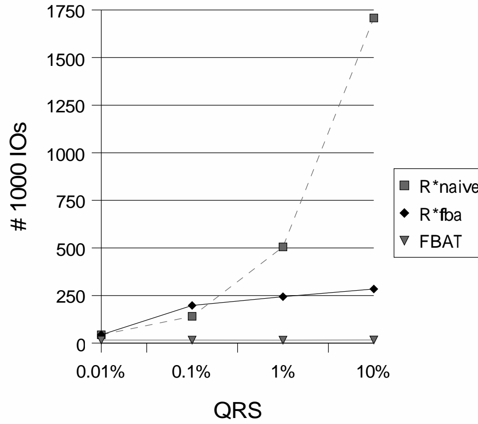


Figure 9: Comparison of the functional box aggregation performance.

7 Conclusion

We examined the problem of computing range-sum queries over objects with non-zero extents in multidimensional space. This 'box aggregation' is a novel problem. Previous database research has concentrated on objects with zero extent (points) on a multidimensional grid, or one-dimensional intervals. We proposed a tree-structured, disk-based, dynamically updated index, the BA tree, that can efficiently answer such queries. We also examined a variation of the problem, the functional box aggregation. Here object values are not constant but are described by functions. This implies that an object's participation in the aggregation result is the function integral over the intersection between the object's box and the query box. To the best of our knowledge this is the first work addressing

functional aggregates. We showed that the BA-tree can be used to maintain functional aggregates for a quite general class of functions. To justify the efficiency of the BA-tree we presented various experimental results. At the expense of some limited extra space, the BA-tree can offer an order of magnitude faster query time than a straightforward approach that uses a plain R-tree. Moreover, its performance remained independent of the query shape or size.

References

[AAD+96] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan and S. Sarawagi, "On the Computation of Multidimensional Aggregates", *Proc. of VLDB*, pp. 506-521, 1996.

[AE98] P. Agarwal and J. Erickson, "Geometric Range Searching and Its Relatives", *Advances in Discrete and Compututational Geometry*, (B. Chazelle, E. Goodman and R. Pollack eds.), American Mathematical Society, Providence, 1998.

[AGP00] S. Acharya, P. Gibbons and V. Poosala, "Congressional Samples for Approximate Answering of Group-By Queries", *Proc. of SIGMOD*, pp. 487-498, 2000.

[Ben80] "Multidimensional Divide-and-Conquer", *Communications of the ACM* 23(4), pp. 214-229, 1980.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multiversion B-Tree", *VLDB Journal* 5(4), pp. 264-275, 1996.

[BKS+90] N. Bechmann, H. Kriegel, R. Schneider and B. Seeger, "The R* tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of SIGMOD*, pp. 322-332, 1990.

[Cha88] B. Chazelle, "A Functional Approach to Data Structures and Its Use in Multidimensional Searching", *SIAM J. Comput.* 17, pp. 427-462, 1988.

[CI99] C. Chan, Y. E. Ioannidis, "Hierarchical Cubes for Range-Sum Queries", *Proc. of VLDB*, pp. 675-686, 1999.

[GAE00] S. Geffner, D. Agrawal and A. El Abbadi, "The Dynamic Data Cube", *Proc. of EDBT*, pp. 237-253, 2000.

[GAE+99] S. Geffner, D. Agrawal, A. El Abbadi and T. Smith, "Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes", *Proc. of ICDE*, pp. 328-335, 1999.

[GHR+99] J. Gendrano, B. Huang, J. Rodrigue, B. Moon and R. Snodgrass, "Parallel Algorithms for Computing Temporal Aggregates", *Proc. of ICDE*, pp. 418-427, 1999.

[HAM+97] C. Ho, R. Agrawal, N. Megiddo and R. Srikant, "Range Queries in OLAP Data Cubes", *Proc. of SIGMOD*, pp. 73-88, 1997.

[HHW97] J. Hellerstein, P. Haas and H. Wang, "Online Aggregation", *Proc. of SIGMOD*, pp. 171-182, 1997.

[KS98] N. Kline and M. Soo, "Time-IT, the Time-Integrated Testbed", ftp://ftp.cs.arizona.edu/timecenter/time-it-0.1.tar.gz, Current as of August 18, 1998.

[Mat94] J. Matoušek, "Geometric Range Searching", *Computing Surveys* 26(4), pp. 422-461, 1994.

[Meh84] K. Mehlhorn, "Multi-dimensional Searching and Computational geometry", *Data Structures and Algorithms*, vol. 3, Springer-Verlag, Heidelberg, West germany, 1984.

[MLI00] B. Moon, I. Lopez and V. Immanuel, "Scalable Algorithms for Large Temporal Aggregation", *Proc. of ICDE*, pp. 145-154, 2000.

[PS85] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Berlin/Heidelberg, Germany, 1985.

[PST93] B. Pagel, H. Six and H. Toben, "The Transformation Technique for Spatial Objects Revisited", *Proc. of SSD*, pp. 73-88, 1993.

[RKR97] N. Roussopoulos, Y. Kotidis and M. Roussopoulos, "Cubetree: Organization of and Bulk Incremental Updates on the Data Cube", *Proc. of SIGMOD*, pp. 89-99, 1997.

[SFB99] J. Shanmugasundaram, U. M. Fayyad, P. S. Bradley, "Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions", *Proc. of KDD*, pp. 223-232, 1999.

[VW99] J. Vitter and M. Wang, "Approximate Computation of Multidimensional Aggregates of Sparse Data using Wavelets", *Proc. of SIGMOD*, pp. 193-204, 1999.

[VWI98] J. Vitter, M. Wang and B. Iyer, "Data Cube Approximation and Histograms via Wavelets", *Proc. of ACM CIKM*, pp. 96-104, 1998.

[YW01] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates", to appear in *Proc. of ICDE*, 2001. (also available at http://www-db.Stanford.EDU/~junyang/research/pubs.html)

[ZDN97] Y. Zhao, P. Deshpande and J. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates", *Proc. of SIGMOD*, pp. 159-170, 1997.

[ZMT+01] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos and B. Seeger, "Efficient Computation of Temporal Aggregates with Range Predicates", to appear in *Proc. of PODS*, 2001. (also listed as *TimeCenter Tech Report 52*)