TITLE PAGE

Submitted PODS'02, paper ID: 31

Efficient Aggregation over Objects with Extent

Donghui Zhang* Vassilis J. Tsotras† Dimitrios Gunopulos‡

Extended Abstract

We examine the problem of efficiently computing sum/count/avg aggregates over objects with non-zero extent. Recent work on computing multi-dimensional aggregates has concentrated on objects with zero extent (points) on a multi-dimensional grid, or one-dimensional intervals. However, in many spatial and/or spatiotemporal applications objects have extent in various dimensions, while they can be located anywhere in the application space. The aggregation predicate is typically described by a multi-dimensional box (*box-sum* aggregation). We examine two variations of the problem. In the simple case an object's value contributes to the aggregation result as long as the object intersects the query box. More complex is the *functional box-sum* aggregation introduced in this paper, where objects participate in the aggregation proportionally to the size of their intersection with the query box. We first show how the simple and the functional box-sum aggregations can be reduced to *dominance-sum* queries. Traditionally, dominance-sum queries are addressed in main memory by a static structure, the ECDF-tree. We then propose two extensions, namely, the ECDF-B-trees, that make this structure disk-based and dynamic. The ECDF-B^q-tree has faster query performance while the ECDF-B^u-tree has better update. Finally, we introduce the BA-tree, an easily implementable, disk-based and dynamic structure for dominance-sum computation, that combines the advantages from each ECDF-B-tree. We run experiments comparing the performance of the ECDF-B-trees, the BA-tree and a traditional R*-tree (which has been augmented to include aggregation information on its index nodes) over spatial datasets. Our evaluation reaffirms that the BA-tree has more robust performance than the ECDF-B-trees. Moreover, when compared against the augmented R*-tree approach, the BA-tree offers drastic improvement in query performance at the expense of some limited extra space.

*CONTACT AUTHOR, Computer Science Department, University of California, Riverside, CA 92521. Tel: 909-787-2838. Fax: 909-787-4643. donghui@cs.ucr.edu

†Computer Science Department, University of California, Riverside, CA 92521. tsotras@cs.ucr.edu This work was partially supported by NSF (IIS-9907477, EIA-9983445) and the Department of Defense.

‡Computer Science Department, University of California, Riverside, CA 92521. dg@cs.ucr.edu This work was partially supported by NSF CAREER Award 9984729, NSF IIS-9907477, the DoD and a gift from AT&T.

1 Introduction

The general *box aggregation problem* is concerned with the computation of aggregation queries over objects with non-zero extents in d-dimensional space. Formally, it is defined 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*". Previous work on multi-dimensional aggregations [16, 27, 10, 14, 26, 24, 13] considers only point objects (i.e., objects with zero extent in all dimensions) that fall on a fixed multi-dimensional grid. One exception is the work in [28, 30] that examines aggregations over interval (i.e., 1-dimensional) objects. 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. [1, 17, 22, 29, 3].

In this paper we address *box-sum* aggregations, i.e. *summation* related aggregations like SUM, COUNT and AVG. What is important for these problems is the number of *extensional* dimensions, i.e., the dimensions over which the data objects can have extent. In most applications there are 2-3 extensional dimensions which are usually the spatial and temporal dimensions of the objects. While our solutions can be easily generalized to the case where all d dimensions are extensional, we concentrate the discussion on spatial and spatiotemporal data. For example, consider a database in an agricultural agency that keeps track of pesticide usage. Each record represents the treatment of an area over a certain time period and contains a 3-dimensional rectangle (that is, a 2-dimensional area describing the field which is sprayed and the corresponding time interval) and a value (the volume of the pesticide). An example of a box-sum query is: "*find the volume of pesticide in Orange County for March 1999*".

We examine two variations of the box-sum aggregation problem. In its simpler form, an object's value contributes to the aggregation result as long as the object intersects the query box. That is, an object's value contributes to the query result as a whole or not at all. Objects that slightly intersect the query box participate in the result with equal importance as objects that are fully contained in the box.

There are applications where the object participation needs to be proportional to the size of the object's intersection with the query box. Furthermore, the value associated with each object can be a function rather than a single constant, which provides for more expressive queries. In the pesticide example, the value associated with each spray record may denote the volume per square yard while the aggregation query asks for the total volume sprayed over a given area. This is a novel problem, namely, the *functional box-sum* aggregation.

A straightforward approach to solve box-sum queries is to index the data objects with a multi-dimensional access method like the *R-tree* [6] and reduce the problem to a *range* search. The aggregate is then computed by identifying the objects that intersect the query box and accumulating their values incrementally. Unfortunately, the performance of this approach is based on how many objects are in the query box, which can be large. Recently, [18, 21] proposed to add aggregation summaries on the R-tree nodes (the aggregate R-tree, or aR-Tree) so as to reduce the number of R-tree nodes visited. Even with this optimization the query effort is still affected by the size of the query box.

We instead propose a different approach that uses specialized aggregate indices. Such indices incrementally maintain aggregates and offer drastic query performance over traditional, object-indexing schemes [28, 30]. We first provide a new approach to reduce the simple box-sum problem to computing *dominance-sums*. This reduction is provably more efficient than previous approaches [12]. Furthermore, we show that for a large collection of functions (polynomials of constant degree), the functional box-sum problem is also reduced to dominance-sums.

The best existing scheme to compute dominance-sums is the ECDF-tree [4], which however is a static, main-memory structure. We propose two extensions to it (the ECDF-B-trees) that make this structure disk-based and dynamic. In particular, the ECDF-B^q-tree guarantees fast query performance while the ECDF-B^u-tree has better update. Then we introduce a novel disk-based and dynamic index, the *Box Aggregation Tree (BA-tree)*, which combines the advantages from each ECDF-B-tree. Experimental results show that the BA-tree has very good average case behavior. It is easily implementable and has more robust performance than the ECDF-B-trees. Furthermore, depending on the sizes of the query boxes we have witnessed an order of magnitude improvement in query performance of the BA-tree against the aR-tree.

We note that our solution applies also to computing *range-sums* over datacubes. The best known solutions for data cube range-sum appear in [13, 9]. When applied to this problem, the BA-tree differs from [13] in two ways. First, it is disk-based, while [13] presents a main-memory structure. Second, the BA-tree partitions the space based on the data distribution while [13] does partitioning based on a uniform grid.

To summarize, the main contributions of this paper are:

1. We provide a new approach to reduce a simple box-sum aggregation query to dominance-sum queries.

2. We introduce a novel variation, the functional box-sum problem and show how it is reduced to dominance-sum queries for a large class of value functions. To the best of our knowledge, this is the first work that addresses the functional aggregation problem.

3. To solve dominance-sum queries, we propose two approaches that transform the main-memory, static ECDF-tree to external structures that can handle dynamic updates (the ECDF-B-trees).

4. We finally propose the BA-tree, which is a dynamic, easily implementable, disk-based index that combines the advantages of both ECDF-B-trees. We provide experimental results to validate the efficiency of the proposed index.

The rest of the paper is organized as follows. Sections 2 and 3 discuss the simple and (respectively) the functional box-sum aggregations and their reduction to dominance-sum queries. Section 4 summarizes the two ECDF-tree extensions, while section 5 introduces the BA-tree. Results from our experimental comparisons appear in section 6. Related work is discussed in section 7 while section 8 provides conclusions.

2 The Simple Box-Sum Problem and Its Reduction

We differentiate between two types of objects: *point* and *box* objects. Given two d-dimensional points $x = (x_1, \cdots, x_d)$ and $y = (x_1, \cdots, y_d)$, we say that x *dominates* y if for every $i \in \{1, \cdots, d\}$, $x_i \geq y_i$. A d-dimensional box b can be described by two corner points: a *low point* which is dominated by all points in the box and a *high point* which dominated all points in the box. The d-dimensional space is itself a box whose low point and high point are represented as p_{min} and p_{max}, respectively. Each object has a value which is used for the aggregation. In the following discussion we refer to the following aggregations:

- **(simple) box-sum:** given a collection S_b of box objects and a query box q, compute SUM$\{o.value | o \in S_b$ and $o.box$ intersects $q\}$;
- **range-sum:** given a collection S_p of point objects and a query box q, compute SUM$\{o.value | o \in S_p$ and $o.point$ is contained in $q\}$;
- **dominance-sum:** given a collection S_p of point objects and a query point p, compute SUM$\{ o.value | o \in S_p$ and $o.point$ is dominated by $p\}$;

The simple box-sum problem is the most general since, (i) the range-sum problem is a special case of the box-sum when the box of each object reduces into a point, and, (ii) the dominance-sum problem is a special case of the range-sum problem with query box $q = (p_{min},\ p)$. The COUNT aggregation problems (*box-count*, *range-count* and *dominance-count*) are special cases of the SUM aggregations, when the value of every object is 1.

[12] proposed a technique to reduce a (simple) box-sum query into a set of dominance-sum queries. Important in the reduction technique is the number of dominance-sum queries a box-sum query is reduced to. The detailed reduction is described in [12] for the 1- and 2-dimensional cases; however, no analysis is presented for the general d-dimensional case. In theorem 1 we show the complexity of the general d-dimensional reduction. Then in theorem 2 we present a new reduction technique that leads to a smaller number of dominance-sum queries. Due to space limitations the theorem proofs are included in appendix A and B.

Theorem 1 *The method of [12] reduces a d-dimensional box-sum query into* $\Omega\left(3^d/\sqrt{d}\right)$ *dominance-sums.*

We now present a new technique which reduces a d-dimensional box-sum query to exactly 2^d dominance-sum queries. Even for small d values our reduction provides a drastic improvement over [12]. For example, with $d = 3$ a method based on [12] would need 26 queries while our technique only 8.

Theorem 2 *A d-dimensional box-sum query is reduced to 2^d dominance-sum queries.*

Intuitively, a d-dimensional box has 2^d corners. An index is maintained for each given corner (e.g. the upper-right corner) of all the objects. A box-sum query is then reduced to 2^d dominance-sum queries, one for each corner of the query box. As an example, consider figure 1 in the 2-dimensional space. Figure 1a shows a query box (with thick border) and two objects intersecting with it. The box-sum query computes the total value of these two objects. In order for a box b to intersect the query box q, the lower left corner of b has to be dominated by the upper right corner of q. Figure 1b shows the candidate boxes. Some candidates are false hits since they are either completely to the left, or completely under, q. The false hits under q correspond to those boxes whose upper left corners are dominated by the lower right corner of q (figure 1c). The false hits to the left of q correspond to those whose lower right corners are dominated by the upper left corner of q (figure 1d). Note that after these false hits are subtracted from the query result, the boxes whose upper right corners are dominated by the lower left corner of q (figure 1e) are subtracted twice. So the total value of them must be added again.

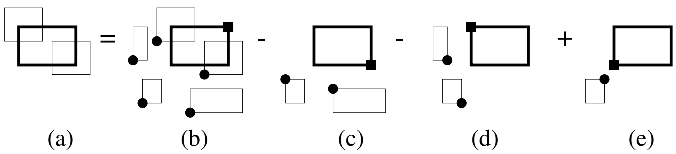
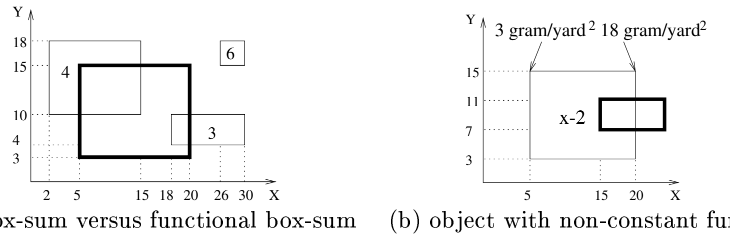


Figure 1: A 2-dimensional box-sum query is reduced to 4 dominance-sum queries

3 The Functional Box-Sum Problem and Its Reduction

In various applications, we are interested in a different kind of box-sum query which we will refer as *functional box-sum.* Figure 2a shows three box objects (with values 3, 4 and 6) and a query. The result of a simple box-sum query in this case is 7, since there are two objects intersecting the query box with values 3 and 4. Consider another scenario where the value associated with each object designates the pesticide volume per square yard while the query asks "*what is the total volume sprayed in the query area*". In this case, the query result is $4 * 50 + 3 * 12 = 236$. Here 50 and 12 are the areas of the intersection between the query box and the two objects, respectively.



(a) box-sum versus functional box-sum (b) object with non-constant function

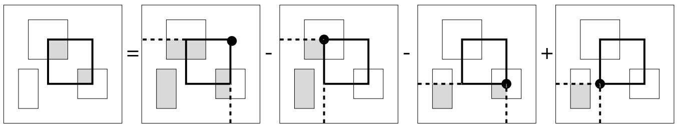
Figure 2: The functional box-sum problem.

In general, the value associated with an object may be a function. While figure 2a contains objects with constant functions, an example of object with non-constant function appears in figure 2b. Here the pesticide was not sprayed evenly over the whole field. At the left border of the field ($x = 5$), it was sprayed 3 grams per square yard. The spray amount increases gradually as x becomes larger until the right border ($x = 20$) where it was sprayed 18 grams per square yard. We can capture this fact by assigning a value to this object that is a function of x (in particular, $f(x, y) = x - 2$). For the query box in figure 2b, the total volume of pesticide as contributed by this object is $(11 - 7) \int_{15}^{20} (x - 2)dx = 310$. Figure 2 reveals that to answer a functional box-sum query, we need (i) the value function of each object, (ii) the area of the object's intersection with the query box, as well as where this intersection is. Assume that the query box in figure 2b is moved leftwards such that it now intersects with the left border of the object but maintains the same size of intersection with the object. The query result would instead be $(11-7)\int_5^{10}(x-2)dx = 110$.

Definition. The *functional box-sum problem* is defined as: "*given a set of objects, each having a box and a value function, and a query box q, compute the total value of all objects that intersect q, where the value contributed by an object r is the integral of the value function of r over the intersection between r and q*".

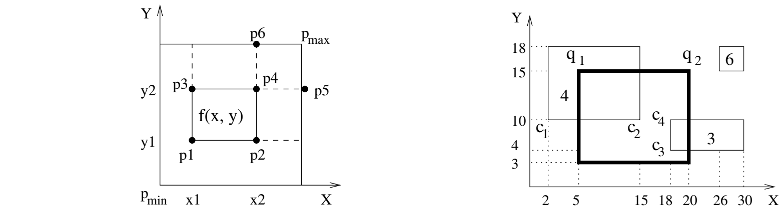
Theorem 3 *For value functions that are polynomials of constant degree, the d-dimensional functional box-sum query is reduced to 2^d dominance-sum queries.*

Proof Sketch. An important special case of the functional box-sum problem is when the low point of the query box is fixed at p_{min}. This is the *origin-involved functional box-sum (OIFBS)*. An OIFBS query is specified by a single point, the high point of the query box. A d-dimensional functional box-sum query can be reduced to 2^d OIFBS queries. Figure 3 depicts an example in two dimensions. Given a query box q, the functional box-sum (figure 3a) is equal to the OIFBS at the upper right corner of q (figure 3b) minus the OIFBS at the upper left corner of q (figure 3c) minus the OIFBS at the lower right corner of q (figure 3d) plus the OIFBS at the lower left corner of q (figure 3e).



(a) (b) (c) (d) (e)

Figure 3: A 2-dimensional functional box-sum query is reduced to 4 OIFBS queries



(a) update in the hypothetical OIFBS index (b) query processing

Figure 4: The functional box-sum problem.

To solve the OIFBS problem, we need to devise an index that logically stores a value for every position in the space. This value is the OIFBS at this position. Suppose such an index exists. Let's consider the effect of inserting a new object into this index. Figure 4a illustrates an object with box $\langle p_1,\ p_4\rangle$ and value function $f(x,y)$. The effect of inserting this object in the hypothetical index contains four parts: (1) for every point $(x,\ y)$ in box $\langle p_1,\ p_4\rangle$, add $\int_{x_1}^{x}\int_{y_1}^{y} f(x',y')\mathrm{d}y'\mathrm{d}x'$; (2) for every point $(x,\ y)$ in box $\langle p_2,\ p_5\rangle$, add $\int_{x_1}^{x_2}\int_{y_1}^{y} f(x',y')\mathrm{d}y'\mathrm{d}x'$; (3) for every point $(x,\ y)$ in box $\langle p_3,\ p_6\rangle$, add $\int_{x_1}^{x}\int_{y_1}^{y_2} f(x',y')\mathrm{d}y'\mathrm{d}x'$; and (4) for every point $(x,\ y)$ in box $\langle p_4,\ p_{max}\rangle$, add $\int_{x_1}^{x_2}\int_{y_1}^{y_2} f(x',y')\mathrm{d}y'\mathrm{d}x'$. The combined effect is equivalent to:

- for every point $(x,\ y)$ in box $b_1=\langle p_1,\ p_{max}\rangle$, add $v_1(x,y)=\int_{x_1}^{x}\int_{y_1}^{y} f(x',y')\mathrm{d}y'\mathrm{d}x'$;
- for every point $(x,\ y)$ in $b_2=\langle p_2,\ p_{max}\rangle$, add $v_2(x,y)=\int_{x_1}^{x_2}\int_{y_1}^{y} f(x',y')\mathrm{d}y'\mathrm{d}x' - \int_{x_1}^{x}\int_{y_1}^{y} f(x',y')\mathrm{d}y'\mathrm{d}x'$;
- for every point $(x,\ y)$ in $b_3=\langle p_3,\ p_{max}\rangle$, add $v_3(x,y)=\int_{x_1}^{x}\int_{y_1}^{y_2} f(x',y')\mathrm{d}y'\mathrm{d}x' - \int_{x_1}^{x}\int_{y_1}^{y} f(x',y')\mathrm{d}y'\mathrm{d}x'$;
- for every point $(x,\ y)$ in $b_4=\langle p_4,\ p_{max}\rangle$, add $v_4(x,y)=\int_{x_1}^{x}\int_{y_1}^{y} f(x',y')\mathrm{d}y'\mathrm{d}x' + \int_{x_1}^{x_2}\int_{y_1}^{y_2} f(x',y')\mathrm{d}y'\mathrm{d}x' - \int_{x_1}^{x}\int_{y_1}^{y_2} f(x',y')\mathrm{d}y'\mathrm{d}x' - \int_{x_1}^{x_2}\int_{y_1}^{y} f(x',y')\mathrm{d}y'\mathrm{d}x'$;

Let $f(x,y)$ be a polynomial of degree at most k. Then, $f(x,y)=\sum_{i=1}^{k} f_i(x,y)$, where $f_i(x,y)=a_i x^{p_i} y^{q_i}$ and $p_i+q_i\le k$. It easily follows that each $v_i(x,y)$ is a polynomial of degree at most $(k+d)$. For instance,

$$v_1=\int_{x_1}^{x}\int_{y_1}^{y}\sum_{i=1}^{t} a_i x'^{p_i} y'^{q_i}\mathrm{d}y'\mathrm{d}x' = \sum_{i=1}^{t}\frac{a_i}{(p_i+1)(q_i+1)}(x^{p_i+1}-x_1^{p_i+1})(y^{q_i+1}-y_1^{q_i+1}),$$

whose degree is $p_i + 1 + q_i + 1 \leq k + 2$. Hence each value function can be represented in *constant* space as a tuple storing its coefficients. At worst there are $O(k^d)$ coefficients, where k and d are constants (in practice this is much less since not all coefficients are present).

Adding value $v_i(x, y)$ to each point in b_i is prohibitively expensive. Instead observe that all b_i's share the same high point p_{max}. Consequently, an update regarding b_i affects all those OIFBS queries specified at points which dominate the p_i. This implies that each such update can be implemented as a *single* point insertion of a value function: we actually insert the point p_i together with the tuple of coefficients of the function $v_i(x, y)$. For instance, the first update regarding b_1 corresponds to: "insert at point p_1 the *tuple* representing the value function $v_1(x, y)$". An OIFBS query specified at point p is then computed by (1) finding the aggregated function over all points dominated by p, and, (2) evaluating the function at p. Since value functions are stored as coefficient tuples, value evaluation is straightforward. As a result, an OIFBS query is reduced to computing a dominance-sum, with the difference that now we store and manipulate value functions instead of single values. □

As an example, consider figure 4b. To compute the OIFBS at point q_1, we find the corner points dominated by q_1 (in this case only c_1) and evaluate their value function at q_1. Here, $c_1 = (2, 10)$ is one of the corner points of the object with value 4. The tuple that has been inserted at c_1 is: $\langle 4, -40, -8, 80\rangle$, since its value function is: $\int_2^x \int_{10}^y 4 \mathrm{d}y' \mathrm{d}x' = 4xy - 40x - 8y + 80$. Evaluating this function at $q_1 = (5, 15)$ yields: $4 * 5 * 15 - 40 * 5 - 8 * 15 + 80 = 60$. Similarly, query point $q_2 = (20, 15)$ dominates four corner points: c_1, c_2, c_3, c_4. The tuples associated at c_2, c_3, c_4 are $\langle -4, 40, 60, -600\rangle$, $\langle 3, -12, -54, 216\rangle$ and $\langle -3, 30, 54, -540\rangle$, respectively. The aggregate of all four tuples is $\langle 0, 18, 52, -844\rangle$ and the result of an OIFBS at q_2 is $18 * 20 + 52 * 15 - 844 = 296$. Note that the OIFBS of the other two corners of the query box are both 0, since they do not dominate any object corner point. The functional box-sum of the query box shown in figure 4b is thus computed as $296 - 60 = 236$, which is the same as our observation in figure 2a at the beginning of this section.

Discussion: In general, we could use functions that (i) are easily aggregated using + and − operators, (ii) can be represented in constant space, and, (iii) can be easily evaluated. While both the simple box-sum and the functional box-sum are reduced to 2^d dominance-sum queries, there are various differences between the two approaches: (i) in the functional dominance-sum problem we maintain tuples instead of single values; (ii) given an index structure that computes dominance-sums the simple box-sum problem needs to maintain 2^d such indices, while the functional box-sum only one; (iii) in the simple box-sum problem inserting a new object corresponds to one update on each of the 2^d indices, while it causes 2^d updates to the single index of the functional box-sum problem.

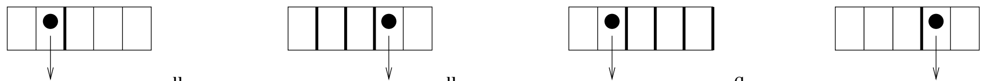
There is an inherent distinction between the simple and the functional box-sum problems. In the functional problem, the value (function) of an object contributes to the query proportional to the intersection size between the object and the query box. If a specialized index is used to solve the functional box-sum problem, this index cannot be used for computing simple box-sums. Conceptually, such index maintains the answer to every possible query box as a weighted sum of the objects that intersect it. The weights correspond to the object intersection sizes with the query box. However, such index does not maintain the actual objects. As a result these weights cannot be de-allocated to their respective objects, making it impossible to answer simple box-sum queries.

4 The ECDF-B-tree

As with box-sums, a dominance-sum query at p can be reduced to *range-reporting* (i.e. "find all the points in the range from p_{min} to p"); this again is inefficient when many points fall in the query range. [4] proposed the ECDF-tree, a main-memory, static data structure to solve this problem.

The ECDF-tree is a *multi-level data structure*, where each level corresponds to a different dimension. At the first level (also called *main branch*), the d-dimensional ECDF-tree is a full binary search tree whose leaves store the data points, ordered by their position in the first dimension. Each internal node of this binary search tree stores a *border* (defined next) for all the points in the left sub-tree. The *border* is itself a (d-1)-dimensional ECDF-tree; here points are ordered by their positions in the second dimension. The collection of all these border trees forms the second level of the structure. Their respective borders are (d-2)-dimensional ECDF-trees (using the third dimension and so on). To answer a dominance-sum query for point $p = (p_1, \ldots, p_d)$, the search starts with the root of the first level ECDF-tree. If p_1 is in the left sub-tree, the search continues recursively on the left sub-tree. Otherwise, two queries are performed, one on the right sub-tree and the other on the border; the respective results are then added together.

To extend the ECDF-tree to handle dynamic updates and disk storage capabilities, we extend the binary search tree at each level into a B+-tree. We call the extended structure the *ECDF-B-tree*. Due to space limitations we only describe the main ideas of the ECDF-B-tree. More details can be found in appendix C. While each internal node of the ECDF-tree has two children, an internal node of the ECDF-B-tree has between $B/2$ and B children. Children are divided by borders. Depending on the meaning of the borders, we have two different versions of the ECDF-B-tree. One version has more efficient update (the *ECDF-B^u-tree*), while the other has better query time (the *ECDF-B^q-tree*).



(a) update of ECDF-B^u-tree (b) query of ECDF-B^u-tree (c) update of ECDF-B^q-tree (d) query of ECDF-B^q-tree

Figure 5: Differences between the two ECDF-B-trees.

The two approaches are illustrated in figure 5, which shows an internal node. Consider a node with children $e_1, e_2, ..., e_m$ and corresponding borders $t_1, t_2, ..., t_{m-1}$. In the ECDF-B^u-tree each border t_i maintains a structure that contains all points in subtree(e_i). In contrast, the structure of a border t_i in the ECDF-B^q-tree contains the points from all subtrees to its left: subtree(e_1), ..., subtree(e_i). To insert a new point into the ECDF-B^u-tree we start from the root node and follow the child e_i that contains this point. This insertion also affects a single border in the root node, border t_i (figure 5a). However, a query needs to examine the subtree that contains the query point q and all borders to the left of q (figure 5b). An insertion in the ECDF-B^q-tree, on the other hand, affects multiple borders (figure 5c) while a query examines only one border (figure 5d) besides the subtree. The above differences affect the I/O performance as explained in theorem 4 (the proof appears in appendix D).

Theorem 4 *The space, bulk-loading, query and update cost of the two ECDF-B-trees are given in table 1. Here n is the number of indexed points, B is the page capacity in number of records, and d is the number of dimensions.*

	Space	Bulk-loading	Query	Update (amortized)
ECDF-B^u-tree	$O\left(\frac{n}{B}\log_B^{d-1} n\right)$	$O\left(\frac{n}{B}\log_B^{d} n\right)$	$O\left(B^{d-1}\log_B^{d} n\right)$	$O\left(\log_B^{d} n\right)$
ECDF-B^q-tree	$O\left(nB^{d-2}\log_B^{d-1} n\right)$	$O\left(nB^{d-2}\log_B^{d} n\right)$	$O\left(\log_B^{d} n\right)$	$O\left(B^{d-1}\log_B^{d} n\right)$

Table 1: ECDF-B-tree Complexity

Clearly, the ECDF-B^q-tree optimizes the query time at the expense of more space and update. The BA-tree presented next attempts to combine the query performance of the ECDF-B^q-tree with the update/space complexity of the ECDF-B^u-tree.

5 The BA-tree

While an ECDF-B-tree is based on the B+-tree the BA-tree is based on the k-d-B-tree [23]. Figure 6 shows an index node of a BA-tree in the 2-dimensional space. As in the k-d-B-tree, each index record is associated with a *box* and a *child* pointer. The boxes of records in a node do not intersect and their union creates the box of the node. Since a node is implemented as a page, we use these terms interchangeably. Each record r points to a sub-tree containing points which are contained in $r.box$.

The goal of augmenting each index record with some extra information is that a dominance-sum query can be answered by following a single sub-tree. Suppose in figure 6a, there is a query point contained in the box of record F. The points that may affect the dominance-sum query of a query point in $F.box$ are those dominated by the upper-right point of $F.box$. Such points belong in four groups: (1) the ones contained in $F.box$; (2) points dominated by the low point of F (in the shadowed region of figure 6a); (3) points below the lower edge of $F.box$ (figure 6b); and (4) points to the left of the left edge of $F.box$ (figure 6c).

To compute the dominance-sum for points in the first group, a recursive traversal of subtree(F) is performed. For points in the second group, we keep in record F a single value called *subtotal*, which is the total value of all these points. For computing the dominance-sum in the third group, we can keep an *x-border* in F which contains the x positions and values of all these points. This dominance-sum is then reduced to a 1-dimensional dominance-sum query for the border. It is then sufficient to maintain these x positions in a 1-dimensional BA-tree. Similarly, for the points in the fourth group, we keep a *y-border* which is a 1-dimensional BA-tree for the y positions of the group's points.

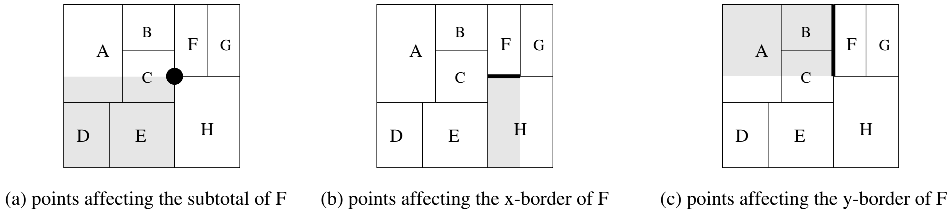


Figure 6: The BA-tree is a k-d-B-tree with augmented border information.

To summarize, the 2-dimensional BA-tree is a k-d-B-tree where each index record is augmented with a single value *subtotal* and two 1-dimensional BA-trees called *x-border* and *y-border*, respectively. The computation for a dominance-sum query at point p starts at the root page R. If R is an index node, it locates the record r in R whose box contains p. A 1-dimensional dominance-sum query is performed on the x-border of r regarding $p.x$. A 1-dimensional dominance-sum query is performed on the y-border of r regarding $p.y$. A 2-dimensional dominance-sum query is performed recursively on page($r.child$). The final query result is the sum of these three query results plus $r.subtotal$.

The insertion of a point p with value v starts at the root R. For each record r where $r.lowpoint$ dominates p, v is added to $r.subtotal$. For each r where p is below the x-border of r, position $p.x$ and value v are added to the x-border. For each record r where p is to the left of the y-border of r, position $p.y$ and value v are added to the y-border. Finally, for the record r whose box contains p, p and v are inserted in the subtree($r.child$). When the insertion reaches a leaf page L, a leaf record that contains point p and value v is stored in L. Since the BA-tree aims at storing only the aggregate information, not the objects themselves, there are chances where the points inserted are not actually stored in the index, thus saving storage space. For instance, if a point to be inserted falls on some border of an index record, there is no need to insert the point into the sub-tree (and eventually stores it in some leaf node) at all. Instead, we simply keep it in the border that contains it. If the point to be inserted falls on the low point of an internal node, there is even no need to inserted it in the border; we simply update the *subtotal* value of the record.

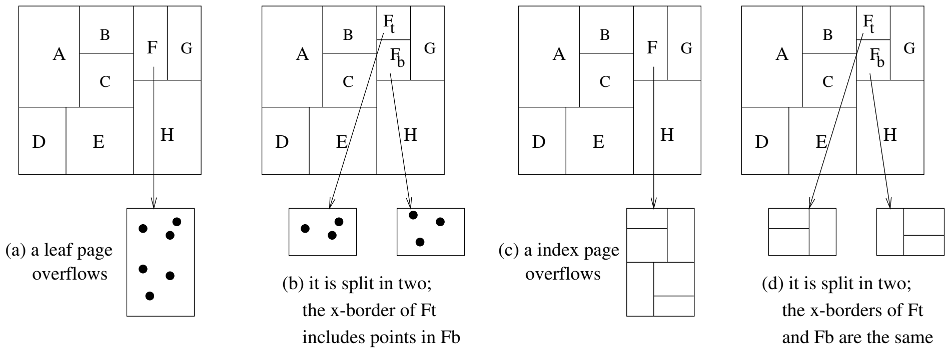


Figure 7: Split in the BA-tree

We next discuss how the insertion algorithm handles overflows. We differentiate between two cases, whether the overflow occurs in a leaf or an index page. Figure 7a shows a leaf page pointed by record F that overflows. The pages is split into two pages F_t and F_b (figure 7b). Since this split is a y-split, the y-border of record F is split in two, one stored at F_t and the other at F_b. The x-border of the bottom record F_b remains the same as that of the previous record F. The x-border of the top record F_t, however, is composed of the x-border of F plus the points in subtree(F_b). Figure 7c considers the case when an index page overflows. The splitting result is shown in figure 7d. Again, the y-border of F is split into two, one for each of the new records. Different from the leaf-split case, however, the x-borders of both F_t and F_b are the same as that of F. To verify the correctness of the split, consider a dominance-sum query where the query point p is contained in $F_t.box$. Obviously, the points in subtree(F_b) should contribute to the query result. However, the x-border of F_t was copied from F and thus does not include any point in the F_b before the split. This is not a problem since the query will recursively go into subtree(F_t), where the border information contains the points in the F_b region inserted before the split.

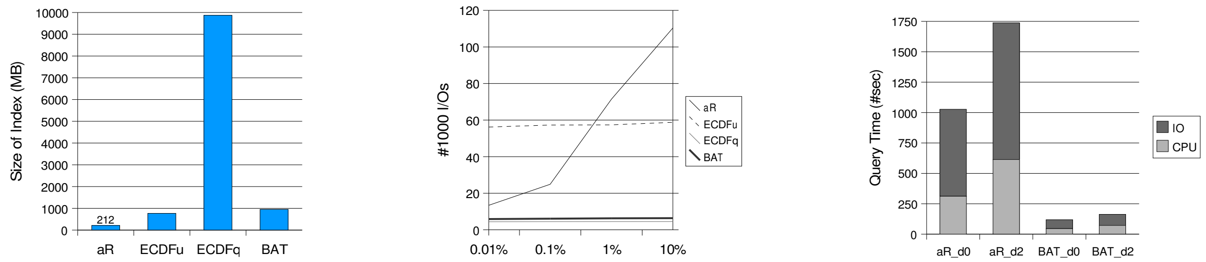
There are various differences between the BA-tree and the ECDF-B-trees. Since the k-d-B-tree is unbalanced, the BA-tree is unbalanced, too. Thus the worst case update and query performance for the BA-tree is linear. However, the average case performance of the BA-tree is much better. For example, if the data points are uniformly distributed, the BA-tree will be rather balanced. To answer a dominance-sum query, a single path in the BA-tree is examined and for each node along the path, a constant number of borders are queried. Thus its average query performance becomes poly-logarithmic (like the ECDF-B^q-tree). As for the update and space complexities, we note that the BA-tree partitions the index page by alternating directions. Thus any line intersecting the box of some index page in a 2-dimensional BA-tree 'cuts' about $\sqrt{B}$ index records. The update of the ECDF-B^u-tree is expensive since each update affects $O(B)$ borders. The BA-tree is faster since only $O(\sqrt{B})$ borders are affected. The space complexity of ECDF-B^q-tree is high because the largest border in a page A contains $O(n_A)$ points, where n_A is the number of points in the sub-tree rooted by A. The BA-tree is better since the largest border contains $O(\frac{n_A}{\sqrt{B}})$ points.

The BA-tree extends to higher dimensions in a straightforward manner: a d-dimensional BA-tree is a k-d-B-tree where each index record is augmented with one *subtotal* value and d borders, each of which is a (d-1)-dimensional BA-tree.

6 Performance

To compare the performance of the various indices we used a dataset with 6 million randomly generated spatial objects in a 2-dimensional space. Each side of an object MBR is on average 1/10,000 of the total dimension size. For solving simple box-sum aggregations, note that we need four dominance-sum indices, we implemented the following: (a) a structure of four ECDF-B^u-trees (this approach is denoted as $ECDF_u$), (b) four ECDF-B^q-trees ($ECDF_q$), and (c) four BA-trees (BAT). We also implemented a plain R*-tree and the aR-tree, which is the R*-tree augmented with aggregate values in index records (denoted as aR). Our initial experiments showed that the BA-tree approach has a query time over 200 times faster than the plain R*-tree approach. So we omit the R*-tree performance and we compare our approaches only with the optimized aR-tree approach. For all indices, we used LRU buffering. For the aR-tree, besides using a LRU buffer, we also used a path buffer which buffers the most recently accessed path of node. We used 8KB page size and 10MB memory buffer.

Figure 8a compares the index sizes. The aR-tree is the smallest index. This is to be expected, since the aR-tree has linear space while the BA-tree and the ECDF-B^u-tree have a logarithmic space overhead. The ECDF-B^q-tree occupies the most space. This is because each update changes too many borders. The BA-tree and the ECDF-B^u-tree have comparable storage requirements, which are much less than the ECDF-B^q-tree. Figure 8b compares the query performance. Each query reports the total number of I/Os spent over 1000 randomly generated query boxes with fixed shape and size. The *query box size (QBS)* is described by the percentage of the query area in the whole space. The aR-tree does not perform well especially when the QBS is large. This is because its worst case performance is linear: most objects in the query box need to be examined. The ECDF-B^q-tree approach performs the best, with the BA-tree being



(a) simple box-sum index sizes (b) simple box-sum query cost varying QBS (c) functional box-sum query cost

Figure 8: Performance comparison.

very close. As expected, the ECDF-B^u-tree has much larger query time since at each index page too many borders need to be checked. Clearly, with the addition of the small space overhead over the aR-tree, the BA-tree approach showed the most robust performance. Moreover, its performance was independent of the query size characteristics (which drastically affects the aR-tree).

We also compared the above indexing schemes for functional box-sum queries. Since the BA-tree was more robust than the ECDF-B-trees we report the comparison between the BA- and aR- trees. Figure 8c depicts the total execution time of 1000 randomly generated queries with QBS being 1% of the space. The execution time is the sum of CPU time (measured by the *getrusage* system call) and the I/O time (measured by the number of I/Os multiplied by 10ms). To observe the impact of different-degree value functions, we used two variations of the original dataset. In the first variation the value of each object was treated as a constant function over the object, i.e., a polynomial of degree zero. In the second variation objects were assigned polynomial functions of degree two. The indices for the degree-zero (degree-two) case use subscript d0 (respectively d2). Clearly, as the degree increases, the query performance worsens since the index becomes larger. Nevertheless, the BA-tree was still drastically faster than the aR-tree. The relative storage requirements (not shown) for the BA-tree and the aR-tree were similar to figure 8a. Due to space limitations, further experimental results appear in the full version of the paper [31].

7 Related Work

We briefly summarize the related work; more details are given in appendix E. The range-reporting query (i.e. find all objects in the query box) has been addressed extensively in the computational geometry field [19, 5, 2]. Most of these works use an optimized version of the *range-tree* originally proposed by [4]. The data cube range-sum problem addresses the following query: given a d-dimensional array A and a query range q, find the sum of values of all cells of A in range q. An initial solution was presented by [16]. The most efficient solutions are the *dynamic data cube* [13] and the *dynamic update cube* [9]. In particular, the dynamic data cube is similar to the ECDF-tree in that it is also a multi-level structure. Also related is the *cumulative* temporal aggregation problem which finds the aggregate value over all records whose intervals intersect a given interval. The most efficient solution to this problem is the *SB-tree* [28]. Recently, [30] introduced the *MVSB-tree* for computing cumulative temporal aggregates with key-range predicates.

8 Conclusions

We considered the problem of computing box-sum queries over objects with extent. Such objects appear in spatial and spatiotemporal applications. We examined two variations, the simple box-sum and the novel functional box-sum problem. In the latter, object values are described by functions and an object's participation in the aggregation result is the function integral over the object's intersection with the query box. To the best of our knowledge this is the first work addressing functional aggregates. Novel reduction techniques were presented to reduce each of these problems to dominance-sum queries. We proposed a tree-structured, disk-based, dynamically-updated index, the BA-tree, that can efficiently answer dominance-sums. We also presented two dynamic, disk-based extensions of computational geometry solutions (the ECDF-B-trees). Experimental results with spatial datasets showed that the BA-tree has the most robust performance. We also compared the BA-tree against the aR-tree, an optimized R-tree whose nodes are augmented with aggregation information. At the expense of some limited extra space, the BA-tree can offer an order of magnitude faster query time. Moreover, the BA-tree query performance is independent of the query shape or size.

Acknowledgements

We would like to thank A. Markowetz for many helpful discussions.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan and S. Sarawagi, "On the Computation of Multidimensional Aggregates", *Proc. of VLDB*, 1996.

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

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

[4] J. L. Bentley, "Multidimensional Divide-and-Conquer", *Communications of the ACM* 23(4), 1980.

[5] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag Berlin Heidelberg, Germany, ISBN 3-540-61270-X, 1997.

[6] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, "The R* tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of SIGMOD*, 1990.

[7] J. L. Bentley and N. B. Saxe, "Decomposable Searching Problems I: Static-to-Dynamic Transformations", *J. of Alg.* 1(4), 1980.

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

[9] C. Chung, S. Chun, J. Lee and S. Lee, "Dynamic Update Cube for Range-Sum Queries", *Proc. of VLDB*, 2001.

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

[11] Y. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry", *Proc. of the IEEE, Special Issue on Computational Geometry*, G. Toussaint (Ed.), 80(9), 1992.

[12] H. Edelsbrunner and M. H. Overmars, "On the Equivalence of Some Rectangle Problems", *Information Processing Letters* 14(3), 1982.

[13] S. Geffner, D. Agrawal and A. El Abbadi, "The Dynamic Data Cube", *Proc. of EDBT*, 2000.

[14] 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*, 1999.

[15] V. Gaede and O. Günther, "Multidimensional Access Methods", *ACM Computing Surveys* 30(2), 1998.

[16] C. Ho, R. Agrawal, N. Megiddo and R. Srikant, "Range Queries in OLAP Data Cubes", *Proc. of SIGMOD*, 1997.

[17] J. Hellerstein, P. Haas and H. Wang, "Online Aggregation", *Proc. of SIGMOD*, 1997.

[18] I. Lazaridis and S. Mehrotra, "Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure", *Proc. of SIGMOD*, 2001.

[19] J. Matoušek, "Geometric Range Searching", *Computing Surveys* 26(4), 1994.

[20] M. H. Overmars, "The Design of Dynamic Data Structures", *LNCS 156*, 1983.

[21] D. Papadias, P. Kalnis, J. Zhang and Y. Tao, "Efficient OLAP Operations in Spatial Data Warehouses", *Proc. of SSTD*, 2001.

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

[23] J. Robinson, "The K-D-B Tree", *Proc. of SIGMOD*, 1981.

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

[25] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data", *to appear in Computing Surveys.* URL= http://www.cs.duke.edu/~jsv/Papers/catalog/node38.html#Vit:I_Osurvey

[26] J. Vitter and M. Wang, "Approximate Computation of Multidimensional Aggregates of Sparse Data using Wavelets", *Proc. of SIGMOD*, 1999.

[27] J. Vitter, M. Wang and B. Iyer, "Data Cube Approximation and Histograms via Wavelets", *Proc. of CIKM*, 1998.

[28] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates", *Proc. of ICDE*, 2001.

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

[30] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos and B. Seeger, "Efficient Computation of Temporal Aggregates with Range Predicates", *Proc. of PODS*, 2001.

[31] D. Zhang, V. J. Tsotras, D. Gunopulos and A. Markowetz, "Efficient Aggregation over Objects with Extent", *TechReport UCR_CS_01_01, CS Dept., UC Riverside*, 2001. URL=http://www.cs.ucr.edu/~donghui/publications/boxaggr_full.ps

Appendix

This appendix contains:

A. Proof of Theorem 1;
B. Proof of Theorem 2;
C. Details of the ECDF-B-tree;
D. Proof of Theorem 4;
E. Detailed Related Work.

A Proof of Theorem 1

We first review the reduction technique of [12] and then show its d-dimensional complexity.

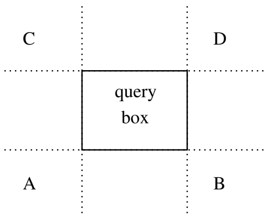


Figure 9: Illustration of the reduction technique of [12].

For simplicity consider the 2-dimensional case depicted in figure 9. Each object is a 2-dimensional rectangle and has a value. To find the sum of values of all objects intersecting a query box, [12] first computes this sum for all objects not intersecting the query box. This is done systematically, by considering objects in the four half-planes described by the boundaries of the query box. For example, the shaded area in figure 9 corresponds to all objects completely above the query box. The sum in this area is computed via a 1-dimensional dominance-sum query, by considering the projection of its objects in the y-axis. Similarly, the sum for objects completely below, to the left and to the right of the query box is computed via 1-dimensional dominance-sum queries. Adding up these four parts is larger than the query result in that objects completely inside one of the areas A, B, C or D have been counted twice. However, the sum in each of these areas can be answered by a 2-dimensional dominance-sum query. For example, an object is in area A if and only if its upper right corner is dominated by the lower left corner of the query box. Hence, a 2-dimensional box-sum query is reduced to four 1-dimensional and four 2-dimensional dominance-sum queries.

Now consider the d-dimensional case. Initially the result $S = 0$. First, for every (d-1)-dimensional *boundary box* (also called *face*) of q, a 1-dimensional dominance-sum query is performed and its result is added to S. A 2-dimensional dominance-sum query is also performed for every (d-2)-dimensional boundary box; these query results are then subtracted from S. If $d \geq 3$, a 3-dimensional dominance-sum query is performed for every (d-3)-dimensional boundary box and the query results are added to S. This process continues until the d-dimensional dominance-sum queries. I.e. for every 0-dimensional boundary box (which corresponds to a corner point of q), a d-dimensional dominance-sum query is performed.

As a result, [12] reduces a d-dimensional box-sum query to a collection of dominance-sum queries, where for every $i \in [1..d]$, the number of i-dimensional dominance-sum queries is equal to the number of (d-i)-dimensional boundary boxes of a d-dimensional box. Since the number of (d-i)-dimensional boundary boxes of a d-dimensional box is $2^i C_d^i$, where $C_d^i = \dfrac{d!}{i!(d-i)!}$, to prove theorem 1, it remains to prove that:

$$\sum_{i=1}^{d} 2^i C_d^i = \Omega\left(\frac{1}{\sqrt{d}} \cdot 3^d\right) \tag{1}$$

Clearly,

$$\sum_{i=1}^{d} 2^i C_d^i \;\; \geq \;\; 2^{2d/3} C_d^{2d/3}$$

From *Stirling's approximation* we have: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$ and thus:

$$\begin{aligned}\sum_{i=1}^{d} 2^i C_d^i \;\; &\geq \;\; 2^{2d/3} \frac{\sqrt{2\pi d}\left(\frac{d}{e}\right)^d\left(1+\Theta\left(\frac{1}{d}\right)\right)}{\sqrt{2\pi\cdot d/3}\left(\frac{d/3}{e}\right)^{d/3}\left(1+\Theta\left(\frac{1}{d}\right)\right)\sqrt{2\pi\cdot 2d/3}\left(\frac{2d/3}{e}\right)^{2d/3}\left(1+\Theta\left(\frac{1}{d}\right)\right)} \\ &= \;\; \Theta\left(\frac{1}{\sqrt{d}}\cdot 3^d\right)\end{aligned}$$

Similarly, we can prove that the number of indices to maintain is also $\Omega\left(\frac{1}{\sqrt{d}}\cdot 3^d\right)$. $\square$

In fact, analysis shows that when $i = 2d/3$, the value for $2^i C_d^i$ is maximal. This implies that the upper bound of the number of queries (indices) is $O\left(\sqrt{d}\cdot 3^d\right)$.

B Proof of Theorem 2

We first introduce some notations. In the rest, q denotes a query box, S the set of objects and $o \in S$ corresponds to an object. We use $(o.l_1, \cdots, o.l_d)$ and $(o.h_1, \cdots, o.h_d)$ to represent the low point and the high point of o's rectangle, respectively. We use the same notation for q. Furthermore, for some dimension i, we define $A_i^0(o,q) \equiv o.l_i < q.h_i$. That is, $A_i^0(o,q)$ is the condition that the low point of object o is dominated by the high point of query box q in the i^{th} dimension. Similarly, we define $A_i^1(o,q) \equiv o.h_i < q.l_i$, which is the condition that the high point of object o is dominated by the low point of query box q in the i^{th} dimension. Object o intersects query q if for every dimension i, the projections of o and q to dimension i intersect. Since the projection of a box to a dimension is an interval and two intervals i_1 and i_2 intersect if and only if $i_1.low < i_2.high$ and not$(i_1.high < i_2.low)$, we can define the box-sum query as:

$$boxsum(S,q) \equiv Sum\left\{o.value | o \in S \wedge \forall i(A_i^0(o,q) \wedge \neg A_i^1(o,q))\right\} \tag{2}$$

Theorem 2 can be inferred by the following lemma which provides a reduction technique from the box-sum problem to 2^d dominance-sums.

Lemma 1 $\quad boxsum(S,q) = \displaystyle\sum_{\forall(s_1,\cdots,s_d),\; s_i\in\{0,1\}} \left((-1)^{\sum_{i=1}^d s_i}\cdot Sum\left\{o.value | o \in S \wedge \bigwedge_{i=1}^{d} A_i^{s_i}(o,q)\right\}\right) \quad (3)$

Discussion: Since each s_i in equation 3 can be either 0 or 1, the set $\{s_1, \ldots, s_d\}$ has 2^d different choices of value assignments. For each choice, the inner summation of equation 3 corresponds to a dominance-sum query. For example, in the 2-dimensional space (figure 1), when $(s_1, s_2) = (1, 0)$, the inner summation is

$$Sum\left\{o.value | o \in S \wedge A_1^1(o,q) \wedge A_2^0(o,q)\right\}.$$

The above summation computes the total value of objects that satisfy two conditions: (a) in dimension 1, the high point of the object is dominated by the low point of the query box; and (b) in dimension 2, the low point of the object is dominated by the high point of the query box. This is equivalent to the total value of objects whose lower right corner is dominated by the upper left corner of the query box (figure 1d). To sum up, a d-dimensional box has 2^d corners. For each specific corner (say the lower right corner), we maintain an index to store this corner of all object boxes. Any box-sum query is reduced to

2^d dominance-sum queries, one per index. A corollary of lemma 1 is theorem 2. It remains to prove the lemma.

Proof of Lemma 1. For clarity, we omit the variables o, S, q from all formulae in the proof. Thus we can re-write equations 2 as:

$$boxsum \equiv Sum\left\{\bigwedge_{i=1}^{d} A_i^0 \wedge \bigwedge_{i=1}^{d} \neg A_i^1\right\}$$

and we need to prove that

$$boxsum = \sum_{\forall(s_1,\cdots,s_d),\ s_i\in\{0,1\}} \left((-1)^{\sum_{i=1}^{d} s_i} \cdot Sum\left\{\bigwedge_{i=1}^{d} A_i^{s_i}\right\}\right).$$

We introduce some functions:

$$B(k) \equiv \begin{cases} \bigwedge_{i=1}^{k} A_i^0, \text{ for } 1 \le k \le d; \\ true, \text{ for } k = 0. \end{cases}$$

$$C(k) \equiv \begin{cases} \bigwedge_{i=1}^{k} \neg A_i^1, \text{ for } 1 \le k \le d; \\ true, \text{ for } k = 0. \end{cases}$$

$$bs(k) \equiv \sum_{\forall(s_k,\cdots,s_d),\ s_i\in\{0,1\}} \left((-1)^{\sum_{i=k}^{d} s_i} \cdot Sum\left\{\bigwedge_{i=k}^{d} A_i^{s_i} \wedge B(k-1) \wedge C(k-1)\right\}\right)$$

Now we identify some properties:

- **Property 1:** $A_i^0 \wedge A_i^1 = A_i^1$;
- **Property 2:** $B(k-1) \wedge A_k^1 = B(k) \wedge A_k^1$;
- **Property 3:** $Sum\{X \wedge \neg Y\} = Sum\{X\} - Sum\{X \wedge Y\}$.

To see the correctness of property 1, consider the meanings of A_i^0 and A_i^1. If (in the i^{th} dimension) the high point of o is dominated by the low point of q, then surely the low point of o is dominated by the high point of q. In other words, we have $A_i^1 \rightarrow A_i^0$ and property 1 holds. We can infer property 2 from property 1, since $B(k) \wedge A_k^1 = B(k-1) \wedge A_k^0 \wedge A_k^1 = B(k-1) \wedge A_k^1$. Regarding property 3, note that given some condition X, $Sum(X)$ means the sum of values of all objects satisfying X. To compute $Sum(X \wedge \neg Y)$, i.e. the value sum of all objects satisfying X but not Y, we can subtract from $Sum(X)$ the value sum of all objects satisfying both conditions X and Y.

We note that $bs(1)$ is equal to the right formula of equation 3. So to prove lemma 1, we will proceed by proving that $bs(d) = boxsum$ and that $\forall k \in [1..d-1]$, $bs(k+1) = bs(k)$. These will infer that $boxsum = bs(1)$, or equation 3.

First, we have:

$$\begin{aligned}
bs(d) &= \sum_{s_d\in\{0,1\}} \left((-1)^{s_d} \cdot Sum\left\{A_d^{s_d} \wedge B(d-1) \wedge C(d-1)\right\}\right) \\
&= Sum\left\{A_d^0 \wedge B(d-1) \wedge C(d-1)\right\} - Sum\left\{A_d^1 \wedge B(d-1) \wedge C(d-1)\right\} \\
&= Sum\left\{B(d) \wedge C(d-1)\right\} - Sum\left\{A_d^1 \wedge B(d) \wedge C(d-1)\right\} \quad \text{(due to property 2)} \\
&= Sum\left\{B(d) \wedge C(d-1) \wedge \neg A_d^1\right\} \qquad \text{(due to property 3)} \\
&= Sum\left\{B(d) \wedge C(d)\right\} \\
&= boxsum
\end{aligned}$$

Now we prove that $\forall k \in [1..d-1],\ bs(k+1) = bs(k)$. By definition,

$$bs(k+1) = \sum_{\forall (s_{k+1},\cdots,s_d),\ s_i \in \{0,1\}} \left((-1)^{\sum_{i=k+1}^{d} s_i} \cdot Sum\left\{ \bigwedge_{i=k+1}^{d} A_i^{s_i} \wedge B(k) \wedge C(k) \right\} \right)$$

The inner summation of the above equation is equal to

$$\begin{aligned}
& Sum\left\{ \bigwedge_{i=k+1}^{d} A_i^{s_i} \wedge B(k) \wedge C(k-1) \wedge \neg A_k^1 \right\} \\
= \; & Sum\left\{ \bigwedge_{i=k+1}^{d} A_i^{s_i} \wedge B(k) \wedge C(k-1) \right\} - Sum\left\{ \bigwedge_{i=k+1}^{d} A_i^{s_i} \wedge B(k) \wedge C(k-1) \wedge A_k^1 \right\} \\
= \; & Sum\left\{ \bigwedge_{i=k+1}^{d} A_i^{s_i} \wedge A_k^0 \wedge B(k-1) \wedge C(k-1) \right\} - Sum\left\{ \bigwedge_{i=k+1}^{d} A_i^{s_i} \wedge A_k^1 \wedge B(k-1) \wedge C(k-1) \right\} \\
= \; & \sum_{s_k \in \{0,1\}} \left((-1)^{s_k} \cdot \bigwedge_{i=k}^{d} A_i^{s_i} \wedge B(k-1) \wedge C(k-1) \right)
\end{aligned}$$

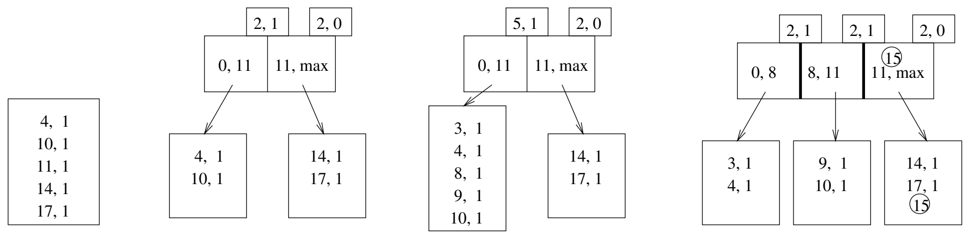
Thus it is easy to see that $bs(k+1) = bs(k)$.
To summarize, we have shown that $boxsum = bs(d) = bs(1) =$ the right formula of equation 3. □

C Details of the ECDF-B-tree

C.1 The One-Dimensional case

In the 1-dimensional case, an ECDF-B-tree is a B+-tree where each index record is augmented with two *border* values (to be explained shortly). As in a B+-tree, there are two types of pages: leaf and index pages. All pages except the root contain between $B/2$ and B records. A leaf record contains a 1-dimensional point and a value. An index record contains a range (*low*, *high*) and a *child* pointer. We augment an index record with two values $border_1$ and $border_2$. They can be implemented in two different ways, resulting to two ECDF-B-tree variations.

The first implementation focuses on optimizing the update time and thus we call the extended structure the *ECDF-B^u-tree*. In this approach, given index record r, the value $r.border_1$ is the total value of points ever inserted in *subtree(r)* (the sub-tree rooted by r), while the value $r.border_2$ is the sum of values of the inserted points with position equal to $r.high$.



(a) After five insertions, the root overflows. (b) The root is split. (c) After three more insertions, the left leaf page overflows. (d) The result tree. A query at 15 affects all borders before.

Figure 10: The update of ECDF-B^u-tree with inserts `11, 14, 4, 10, 17, 9, 3, 8`

To insert a point p with value v, we start with the root page A. If it is an index page, we find the record r whose (low, $high$) contains p; we then add v to $r.border_1$ and recursively insert into $subtree(r.child)$. To insert (p, v) into a leaf page, it is simply recorded into the page. If the page overflows, it is split into two pages. For example, consider figure 10. Assume $B = 4$, i.e. a page can contain between 2 and 4 records.

Also assume that the sequence of points to be inserted is `11, 14, 4, 10, 17, 9, 3, 8` and the value for every point is 1. Initially the root page is empty. After the first five updates, the root page overflows and is split into two, creating a new root (figure 10b). Point (11, 1) does not appear in any of the two leaf pages since it is chosen as the splitting position. Its value is recorded as $border_2$ of the first index record in the new root page. The range of each index record is shown in the larger rectangle and the border values are shown in the smaller rectangle on top. For instance, the range for the first index record in figure 10b is (0, 11) and the two border values of it are 2 and 1, respectively. The next three updates cause an overflow in the left leaf page (figure 10c). The page is split similarly to figure 10b and the result tree is shown in figure 10d.

An interesting observation is that between the two border values 5 and 1 of the split index record, 5 is discarded but 1 is kept as $border_2$ of the new index record on the right. This scenario explains why $border_1$ is kept separately from $border_2$. If an insertion point falls on a border of some index page, only $border_2$ of the corresponding record is modified and no recursive insertion is needed. Thus if (11, 5) is inserted in figure 10d, the only change is that $border_2$ of the second record is changed from 1 to 6. Since we keep the aggregates of the points and not the original points themselves, deleting a point in the base table is treated as an insertion with a negative value. However, this does not necessarily mean that the tree keeps growing. As an example, consider the insertion of (4, -1) in figure 10d. The insertion is directed to the leftmost leaf page and changes a record in it from (4, 1) to (4, 0) which is then removed. Since the page underflows, it is combined with a sibling page.

We now consider processing a dominance-sum query with the ECDF-B^u-tree. In an index page, all borders dominated by the query point contribute to the query result, and the query is recursively processed in one branch of the sub-trees. In a leaf page, to answer the query is trivial. As an example, we show how to find the total value of points dominated by 15 in figure 10d. In the root node, the sum of borders dominated by 15 is $2+1+2+1=6$. In the corresponding sub-tree, there is one point (14, 1) dominated by 15. So the total result is $6+1=7$. Note that if the query point falls on some border, the sub-tree query traversal is not needed. E.g. if the query point is 11, the query result 6 can be computed solely by examining the border values in the root node.

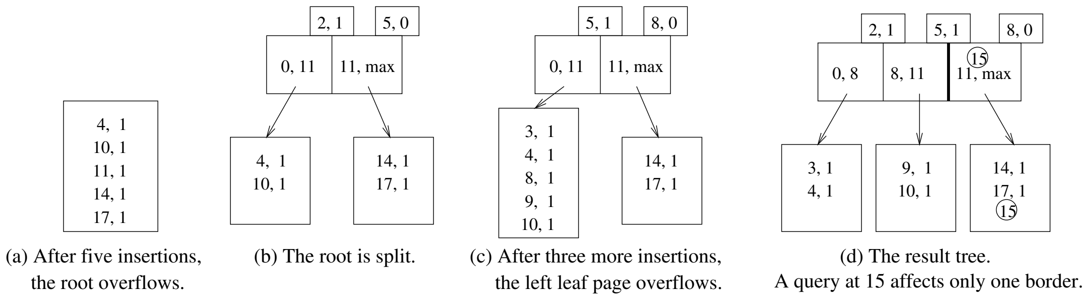


Figure 11: The update of ECDF-B^q-tree

The *ECDF-B^q-tree* focuses on optimizing query time. Given index record r in page A, the value $r.border_2$ has the same meaning as before. But the value $r.border_1$ corresponds to the total value of points ever inserted in A where the point position is less than $r.high$. The insertion algorithm is quite similar to that of the ECDF-B^u-tree. The difference is that to insert $(p,\ v)$ into an index page A, all borders in A whose position is after p should be updated. Using the same update sequence as in figure 10, the evolution of the second approach is shown in figure 11. Note that when the root splits (figure 11b), $border_1$ of the second record is 5 compared with 2 in the previous example. Also note how in figure 11c, the insertion in the left sub-tree changes $border_1$ of the right index record from 5 to 8. To perform a query at 15 in figure 11d, besides the sub-tree containing 15, only one border is examined. The result is $5+1+1=7$, which is the same as in the first approach.

C.2 The d-Dimensional case

For simplicity we start with the 2-dimensional case. As points (x, y) are inserted, a tree structure is built on their x coordinates, similar to the one discussed in the previous section. Each index record still maintains two borders $border_1$ and $border_2$. However, a border points to a 1-dimensional ECDF-B-tree, consisting of the y coordinates of the points inserted. Again, there are two approaches, one with better update and the other with better query performance.

We focus on the ECDF-B^u-tree (the discussion of the ECDF-B^q-tree is similar). Suppose we want to insert a 2-dimensional point $p = (p_1,\ p_2)$ and a value v into an index page A. We first find the record r in A such that $r.high \geq p_1$ and $r.low < p_1$. As shown in figure 5a, if $r.high > p_1$, we recursively insert into subtree($r.child$) and we insert the 1-dimensional point p_2 and value v into the 1-dimensional sub-tree rooted by $r.border_1$. If $r.high = p_1$, i.e. the point falls on some border, we simply insert point p_2 and value v into subtree($r.border_2$). If we focus on the "main branch" of the 2-dimensional tree, (at most) a single path from the root to the leaf level is examined. To perform a query, as shown in figure 5b, the 1-dimensional trees corresponding to all the borders to the left of the query point are queried. Besides, (at most) a single 2-dimensional sub-tree is queried.

The extension to the d-dimensional case is straightforward. In general, a d-dimensional ECDF-B-tree is a B+-tree which indexes the first dimension of the inserted points. Besides the usual child pointer and range (low, $high$), an index record has two borders, each pointing to a (d-1)-dimensional ECDF-B-tree. The ECDF-B-trees can be bulk-loaded. The points are sorted and bulk-loaded into a B+-tree based on one dimension. Since the process of bulk-loading a B+-tree changes only the rightmost path, as each node not on the rightmost path is generated, the border information of the node can be calculated (by bulk-loading a lower-rank ECDF-B-tree, etc.). An implementation issue is that a border may contain only a few points and thus it is wasteful to keep a separate tree for this border (which costs one I/O to retrieve). To avoid this, we can use a single disk page to keep multiple borders, preferably the borders in the same index page.

D Proof of Theorem 4

We use S_u, L_u, Q_u and U_u to represent the space, bulk-loading, query and update complexity of the ECDF-B^u-tree, respectively. Similarly, we use S_q, L_q, Q_q and U_q to represent the complexity of the ECDF-B^q-tree.

Space and bulk-loading complexity: The main branch of any of the ECDF-B-trees occupies $O\left(n/B\right)$ space. The space of the borders dominates the overall space. The ECDF-B^u-tree at level 1 (the root level) has $O(B)$ borders, each of which roots a (d-1)-dimensional tree containing n/B points. In general, at level i, there are $O(B^i)$ borders, each of which is a (d-1)-dimensional tree containing $O(n/B^i)$ points. Thus we have:

$$S_u(n,d) = \frac{n}{B} + \sum_{i=1}^{\log_B n-1} B^i S_u(\frac{n}{B^i}, d-1)$$

The proof uses induction. For $d = 1$, the space complexity holds, since it is basically a B+-tree. Suppose the complexity is correct for (d-1)-dimensional trees. Then for the d-dimensional case,

$$\begin{aligned} S_u(n,d) &= \frac{n}{B} + \sum_{i=1}^{\log_B n-1} B^i \cdot O\left(\frac{n}{B^i}/B \cdot \log_B^{d-2} \frac{n}{B^i}\right) \\ &= O\left(\frac{n}{B} \log_B^{d-1} n\right) \end{aligned}$$

Similarly, we have:

$$L_u(n,d) = \frac{n}{B}\log_B n + \sum_{i=1}^{\log_B n-1} B^i L_u(\frac{n}{B^i}, d-1)$$

and by induction, we can prove that $L_u = O\left(\frac{n}{B}\log_B^d n\right)$.

As for the ECDF-B^q-tree, at level 1, there is 1 node which has $O(B)$ borders. The t^{th} border is a (d-1)-dimensional tree with $O(nt/B)$ points. In general, at level i, there are B^{i-1} nodes, each of which has $O(B)$ borders. For each node, the t^{th} border is a (d-1)-dimensional tree with $O(nt/B^i)$ points. So the space and bulk-loading costs of the ECDF-B^q-tree are:

$$S_q(n,d) = \frac{n}{B} + \sum_{i=1}^{\log_B n - 1} B^{i-1} \sum_{t=1}^{B} S_q\left(\frac{nt}{B^i}, d-1\right)$$

$$L_q(n,d) = \frac{n}{B}\log_B n + \sum_{i=1}^{\log_B n - 1} B^{i-1} \sum_{t=1}^{B} L_q\left(\frac{nt}{B^i}, d-1\right)$$

By induction we can prove that $S_q = O\left(nB^{d-2}\log_B^{d-1} n\right)$ and $L_q = O\left(nB^{d-2}\log_B^d n\right)$.

Query complexity: For both ECDF-B-trees, the query examines a single path in the main branch of the tree, which takes $O(\log_B n)$ I/Os. The major concern is the complexity of querying the borders. For the ECDF-B^u-tree, at every level i, there are $O(B)$ borders to query, each of which is a (d-1)-dimensional tree with $O(n/B^i)$ points. So we have:

$$\begin{aligned} Q_u(n,d) &= \log_B n + \sum_{i=1}^{\log_B n - 1} B Q_u\left(\frac{n}{B^i}, d-1\right) \\ &= O\left(B^{d-1}\log_B^d n\right) \end{aligned}$$

For the ECDF-B^q-tree, at every level i, there is only one border that needs to query. At level i the border is a (d-1)-dimensional tree with $O(n/B^i + 1)$ points. Thus,

$$\begin{aligned} Q_q(n,d) &= \log_B n + \sum_{i=1}^{\log_B n - 1} Q_q\left(\frac{n}{B^{i-1}}, d-1\right) \\ &= O\left(\log_B^d n\right) \end{aligned}$$

Update complexity: At every level i of a ECDF-B^u-tree, if the node does not split, only 1 border needs to be updated. The border is a (d-1)-dimensional tree with $O(n/B^i)$ points. So the update complexity is

$$\begin{aligned} U_u(n,d) &= \log_B n + \sum_{i=1}^{\log_B n - 1} U_u\left(\frac{n}{B^i}, d-1\right) \\ &= O\left(\log_B^d n\right) \end{aligned}$$

Note that the above complexity is acquired by assuming that nodes do not split. If a level i node splits, $O(B)$ borders need to be generated, which is expensive. To bulk-load these borders, it takes $O(n_0 \log_B^d n_0)$ I/Os, where $n_0 = n/B^i$. However, in the B+-tree, if a newly generated node r roots a sub-tree of n_0 leaf records, on average n_0 insertions need to go through r before it splits again. So the cost of bulk-loading the borders due to a split can be amortized to the n_0 insertions. The overall amortized update complexity remains the same.

Similarly, assuming no splits, an update operation in the ECDF-B^q-tree affects B borders at every level i. Out of the B borders, the t^{th} is a (d-1)-dimensional tree which indexes $O(nt/B^i)$ points. So the cost of an update is

$$
\begin{aligned}
U_q(n,d) &= \log_B n + \sum_{i=1}^{\log_B n - 1} \sum_{t=1}^{B} U_q\left(\frac{nt}{B^i}, d-1\right) \\
&= O\left(B^{d-1} \log_B^d n\right)
\end{aligned}
$$

The splits can also be amortized similar to the previous case. □

E Detailed Related Work

Computational Geometry. A range-reporting query reports all objects in the query box [19, 5, 2]. [4] proposed the *range-tree* for the range-reporting query. The range-tree is very similar to the ECDF-tree. The best internal memory solution for the 2-dimensional range-counting problem is given in [8]. For the static case (i.e. all n points are known in advance), the solution uses $O(n)$ space and has $O(\log_2 n)$ query time. In particular, a range-tree is compressed using the *functional approach* technique. To extend the solution to the d-dimensional case, the *multi-dimensional divide-and-conquer* technique of [4] can be used which leads to $O(n \log_2^{d-1} n)$ space and $O(\log_2^{d-1} n)$ query time. However, note that the solution of [8] applies only to range-counting and not to range-sum. Furthermore, the data structure is rather complex to implement in practice.

Note that the ECDF-tree and the range-tree are both static and internal-memory structures. To *dynamize* a static data structure some standard techniques can be used [11]. For example, the *global rebuilding* [20] or the *logarithmic method* [7]. To *externalize* an internal-memory data structure, a widely used method is to augment it with block-access capabilities [25].

Range-Sum for Data Cubes. The data cube range-sum problem addresses the following query: given a d-dimensional array A and a query range q, find the sum of values of all cells in A in range q. [16] proposed to maintain a *prefix-sum array* P which is of the same size as A. The range-sum query is then transformed into 2^d array look-ups in P and their result is combined through additions and subtractions. However this approach uses $O(k)$ update cost, where k is the number of array cells.

The *dynamic data cube* [13] reduces the update cost to $O(\log^d k)$ while it increases the query time from $O(1)$ to $O(\log^d k)$. The idea is to compute each prefix-sum needed on-the-fly. The array A is partitioned into sub-arrays which are then organized into a tree structure. To perform an update or to answer a prefix-sum query, the dynamic data cube examines a single path from the root to a leaf. Like the ECDF-tree, the dynamic data cube is a multi-level structure: within each tree node, a (d-1)-dimensional dynamic data cube is kept to speed-up the query. Recently, [9] proposed the *dynamic update cube* which further improves the update cost to $O(\log k_u)$, where k_u is the number of changed array cells.

Temporal Aggregation. The *cumulative* temporal aggregation query finds the aggregate value over all records whose intervals intersect a given interval. Since a time interval can be regarded as a 1-dimensional box, the cumulative temporal aggregation query for SUM is an 1-dimensional box-sum query. [28] developed efficient structures to incrementally maintain temporal aggregates. Among them, the *JSB-tree* computes cumulative temporal aggregates using $O(\log_B N)$ update and query time, where B is the page capacity and N the total number of updates. Recently, [30] introduced the *MVSB-tree* for computing cumulative temporal aggregates with key-range predicates. This problem is a special case of the 2-dimensional box-sum query. The query time is $O(\log_B N)$ and the update time is $O(\log_B K)$, where K is the number of different keys ever inserted in the index.