Range Aggregate Processing in Spatial Databases

Yufei Tao and Dimitris Papadias

Abstract—A *range aggregate query* returns summarized information about the points falling in a hyper-rectangle (e.g., the total number of these points instead of their concrete ids). This paper studies spatial indexes that solve such queries efficiently and proposes the *aggregate Point-tree* (aP-tree), which achieves logarithmic cost to the data set cardinality (independently of the query size) for two-dimensional data. The aP-tree requires only small modifications to the popular multiversion structural framework and, thus, can be implemented and applied easily in practice. We also present models that accurately predict the space consumption and query cost of the aP-tree and are therefore suitable for query optimization. Extensive experiments confirm that the proposed methods are efficient and practical.

Index Terms—Database, spatial database, range queries, aggregation.

1 Introduction

Traditional research in spatial databases often aims at the *range query*, which retrieves the data objects lying inside (or intersecting) a multidimensional hyper-rectangle. In many scenarios (e.g., statistical analysis, data warehouses, etc.), however, users are interested only in summarized information about such objects, instead of their individual properties. Consider, for example, a spatial database managing the hotels in a city; a query for statistical purposes could ask for the number of hotels in a certain district (rather than their respective locations), or the average rate per night of these hotels. Furthermore, in some applications such as traffic monitoring, results of range queries (e.g., finding all vehicles in the center) are meaningless (due to frequent object movements), while the aggregate information (the number of vehicles) is usually fairly stable (i.e., approximately the same number of vehicles enter and exit the region within a short period of time) and measurable (e.g., through sensors across the region [12]).

Specifically, given a set S of points in the d-dimensional space, a *range aggregate* (RA) query returns a single value that summarizes the set $R \subseteq S$ of points in a d-dimensional hyper-rectangle q according to some aggregation function. Aggregation functions are divided into three classes [18]: distributive, algebraic, and holistic. *Distributive aggregates* (e.g., *count*, *max*, *min*, *sum*) can be computed by partitioning the input into disjoint sets, aggregating each set individually and then obtaining the final result by further aggregating the partial results. Algebraic aggregates can be expressed as a function of distributive aggregates: *average*, for example, is defined as *sum/count*. Holistic aggregates (e.g., *median*), on the other hand, cannot be computed by dividing the input into parts. In this paper, we consider *range count queries* on multidimensional data points, where the result is the size of R (e.g., the number of hotels in an area q), but the solutions apply to any distributive, or algebraic (but not to holistic) aggregates with straightforward adaptation.

- *Y. Tao is with the Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: taoyf@cs.cityu.edu.hk.*
- *D. Papadias is with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: dimitris@cs.ust.hk.*

Manuscript received 9 Apr. 2003; revised 4 Oct. 2003; accepted 10 Nov. 2003. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0031-0403.

1.1 Motivation

A RA query can be trivially answered as an ordinary range query, i.e., by first retrieving the qualifying objects and then aggregating their properties. This, however, incurs significant overhead since, intuitively, the objective is to retrieve only a single value (as opposed to every qualifying object). A common solution is the *aggregate index*, which augments a traditional spatial access method with summarized information in intermediate entries (as elaborated in the next section). The most popular aggregate index is the *aggregate R-tree* (aR-tree) [21], [29], [26], which outperforms traditional R-trees considerably on RA retrieval.

Aggregate processing of multidimensional objects has also been studied theoretically, leading to several interesting results [40], [16], [41]. As explained in the next section, despite their attractive theoretical bounds, the proposed methods have limited practical applicability due to several reasons. First, the asymptotical performance may contain (potentially large) hidden constants, which renders the actual performance of the structure prohibitively expensive in practice. Second, theoretical bounds are insufficient for query optimization, which requires accurate analytical models quantifying the actual space consumption and query cost. Third, the resulting indexes either rely on some restrictive assumptions, or have rather complicated structures and, thus, require considerable implementation efforts.

1.2 Contributions

We first present an asymptotical performance analysis for the aR-tree proving that its average RA query cost is

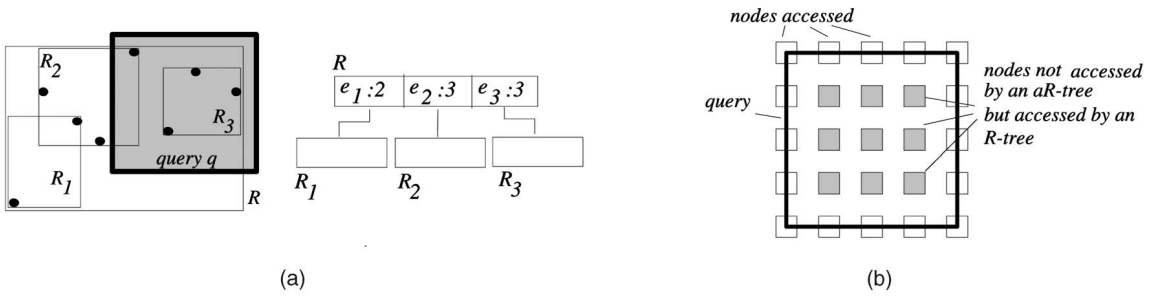

Fig. 1. The aR-tree. (a) An example. (b) Cost savings compared a normal R-tree.

$O((K/B)^{(d'-1)/d'})$, where K is the number of points in the query window, B the disk page size, and d' the *fractal dimensionality* [14] of the data set. It is clear that the cost degrades as the query size increases and eventually reaches $O((N/B)^{(d'-1)/d'})$, where N is the data set cardinality. Motivated by this, we develop a new access method, the *aggregate Point tree* (aP-tree), which achieves logarithmic cost $O(\log_B N)$ for any query on two-dimensional data. The aP-tree requires only small modifications of the well-studied multiversion structure, thus incurring minimum implementation overhead. We also propose algorithms for bulkloading and dynamically maintaining the aP-tree and derive a probabilistic cost model that quantifies its actual space and query overhead. Finally, we show that the combination of the aR and aP-trees leads to an index that solves RA queries in three (or higher) dimensional spaces efficiently.

Our discussion is based on the typical memory-disk hierarchy, where each I/O access transfers a page of B (i.e., the page size) units of information from the disk to the main memory that contains at least B^2 pages (a reasonable assumption in practice). The rest of the paper is organized as follows: Section 2 surveys the existing techniques for evaluating RA queries (focusing on the aR-tree) as well as multiversion structures. Section 3 provides our asymptotical analysis of the aR-tree, identifying its deficiencies, while Section 4 discusses the aP-tree, including the construction/query algorithms and the cost models. Section 5 evaluates the proposed methods through extensive experimental evaluation with synthetic and real data sets, and Section 6 concludes the paper with directions for future work.

2 Related Work

Section 2.1 discusses the aR-tree and its analysis, while Section 2.2 overviews other related approaches. Section 2.3 elaborates the multiversion B-tree which motivates our solutions.

2.1 The Aggregate R-Tree (aR-Tree)

The aR-tree enhances the conventional R-tree [15], [31], [8] by keeping aggregate information in intermediate nodes. Fig. 1a shows an example, where for each intermediate entry, in addition to the minimum bounding box (MBB), the tree stores the number of points in its subtree (i.e., *count* aggregate function). To answer a RA query q (the shaded rectangle), the root R is first retrieved and its entries are compared with q. For every entry, there are three cases: 1) The (MBB of the) entry (e.g., e_1) does not intersect q and, thus, its subtree is not explored further. 2) The entry partially intersects q (e.g., e_2) and its child node is fetched to continue the search. 3) The entry is contained in q (e.g., e_3), in which case we simply add the aggregate number of the entry (i.e., 3 for e_3) without accessing its subtree. As a result, only two node accesses (R and R_2) are necessary, while a conventional R-tree (i.e., without the aggregate numbers) would also visit R_3. The cost savings increase with the size of the query, which is an important fact because in practice RA queries often involve large rectangles (e.g., in OLAP applications, an "all" operator [18] is equivalent to a RA query covering the whole universe).

In order to analyze the performance of the aR-tree let us consider the case where data points and query hyper-rectangles distribute uniformly in the d-dimensional space (without loss of generality, assume that each axis has unit length). Fig. 1b shows the node MBBs of an R-tree indexing uniform points[1] and a query q. The nodes accessed by the query are the ones whose MBB intersects, but is not contained in, the query hyper-rectangle (white rectangles in Fig. 1b). Let $PR_{intr}(q,o)(PR_{cont}(q,o))$ be the probability that q intersects (contains) MBB o. Then, the *access probability* $PR_{acs(q,o)}$ that a node o is visited by a query q can be computed as:

$$PR_{acs}(q,o) = PR_{intr}(q,o) - PR_{cont}(q,o). \quad (1)$$

Given the length o_j of (the MBB of) o on the jth dimension ($1 \le i \le d$) (similarly q_j for q), Kamel and Faloutsos [23] present the following formula for $PR_{intr}(q,o)$:

$$PR_{intr}(q,o) = \prod_{j=1}^{d} (q_j + o_j). \quad (2)$$

$PR_{cont}(q,o)$ is solved by Jurgens and Lenz in [22]:

$$PR_{cont}(q,o) = \begin{cases} \prod_{j=1}^{d} (q_j - o_j) & \text{if } q_j > o_j \text{ for all dimensions } j \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

Combining (1), (2), and (3), the access probability $PR_{acs}(q,o)$ equals:

1. For uniform distribution, nodes at the same level of the aR-tree have square MBBs (i.e., the length of each MBB is identical on each axis) with similar sizes [35].

$$PR_{acs}(q,o) = \begin{cases} \prod_{j=1}^{d}(o_j+q_j) - \prod_{j=1}^{d}(q_j-o_j) & \text{if } q_j > o_j \text{ for all dimensions } j \\ \prod_{j=1}^{d}(o_j+q_j) & \text{otherwise.} \end{cases} \quad (4)$$

Let s_i be the edge length of a MBB at the ith level, $0 \le i \le h-1$, where $h = \log_B(N/f)$ is the tree height and 0 stands for the leaf level. For point data, s_i is given by [35]:

$$s_i = \min\left\{\left(f^{i+1}/N\right)^{1/d}, 1\right\}, \quad (5)$$

where f is the average node fanout (i.e., number of entries in a node), and N the data set cardinality. Since the number of nodes at the ith level is $N/f^{i+1} (0 \le i \le h-1)$, the expected number of node accesses for answering a RA query q is given by:

$$Cost(q) = \sum_{i=0}^{h-1}\left[\left(\frac{N}{f^{i+1}}\right) \cdot PR_{acs-i}(q, s_i)\right], \quad (6)$$

where $PR_{acs-i}(q, s_i)$ is the access probability at the ith level, obtained from (4).

The above results can be extended to nonuniform data. Jurgens and Lenz [22] derive the alternative $PR_{acs}(q,o)$ (i.e., the most important component in the above equation) for three special distributions (but their formulae do not generalize to other distributions). Theodoridis and Sellis [35] propose a technique that integrates (6) with histograms to support arbitrary distributions. Their idea is to sample data statistics around the query region and use them to compute the appropriate N and s_i before evaluating (6) (i.e., different queries lead to distinct N and s_i, and, therefore, different estimates). Faloutsos and Kamel [14] propose another cost model that can be applied to arbitrary data distribution using the fractal dimensionality[2] d' of the data set. The resulting formula is similar to (4), except that all occurrences of d are replaced with d'. The problem of the fractal technique is that it cannot provide a separate cost estimate for each individual query (i.e., it can produce only a single cost corresponding to the average query performance), thus limiting its applicability in practice.

2.2 Other Related Approaches

Based on the *SB*-tree [39], Zhang et al. [40] propose the *MVSB*-tree that efficiently solves a range aggregate query on two-dimensional horizontal interval data (i.e., find the number of intervals intersecting a query rectangle) in $O(\log_B(N/B))$ I/Os, using $O((N/B)\log_B(N/B))$ space. Their idea is to transform a RA query to four "less-key-less-time" and two "less-key-single-time" queries. The MVSB-tree can also answer RA queries on 2D points with the same asymptotical performance by treating each point as a special interval with zero length. The aP-tree proposed in this paper achieves the same time and space complexity using a simpler transformation. Further, the MVSB-tree is applicable only to two-dimensional spaces and its analysis is limited only to asymptotical performance.

Zhang et al. [41] develop two versions of the *ECDF-B-tree* for RA queries on rectangular objects (and hence also data points) with different space-query time trade-offs. Specifically, for d-dimensional data, the first version consumes $O((N/B)\log_B^{d-1}(N/B))$ space and answers a query in $O(B \cdot \log_B^d(N/B))$ I/Os, while the corresponding complexities of the second version are $O(N \cdot B^{d-2}\log_B^{d-1}(N/B))$ (for space) and $O(\log_B^d(N/B))$ (for query cost). These bounds are worse than the MVSB and aP-tree (due to the wider applicability of ECDF-B-trees); furthermore, there is no cost model for practical performance.

Govindarajan et al. [16] present the *CRB-tree* that achieves asymptotically optimal performance (i.e., linear space consumption and logarithmic query time) for RA query processing. The excellent performance of the CRB-tree, however, is based on two stringent assumptions. First, it assumes that an integer with value v is represented by exactly $\log_2 v$ bits so that multiple integers may be compressed into a single word, which significantly increases the implementation overhead. To the best of our knowledge, all the practical indexes (including the proposed aP-tree) adopt the conventional memory model that uses a fixed number of bytes for storing each integer. Further, it is not clear how a float number can be compressed without precision loss. Second, the CRB-tree must be stored in consecutively-addressed pages (i.e., an array), so that the address of a page to be visited can be calculated directly (thus avoiding the cost of traversing parent levels). If this condition is not satisfied (e.g., due to disk fragmentation), its cost increases significantly and eventually the tree becomes as expensive as sequential scan. Finally, the extension of the CRB-tree to higher dimensional spaces (see [16] for details) results in a multilayered structure that is merely of theoretical interest.

Several papers [20], [11], [17], [10] aim at accelerating aggregate queries in "dense" data warehouses where each dimension i $(1 \le i \le d)$ contains v_i discrete values, and the data set cardinality equals (or approximates) $\prod_{i=1\sim d}(v_i)$ (i.e., the cardinality is exponential to the dimensionality). These solutions are not applicable to continuous data axes. Furthermore, even for discrete dimensions, the cardinality of practical data sets is usually much lower (by orders of magnitude) than $\prod_{i=1\sim d}(v_i)$ [37], in which case their space overhead is prohibitive. Finally, related work on multidimensional histograms [27], [28], [3], [19] aims at efficiently obtaining approximate aggregate results (for selectivity estimation). In this paper, we discuss exact RA processing.

2.3 The Multiversion B-Tree (MVB-Tree)

Consider a temporal database, where data are best described as horizontal intervals in the key-time space [32]. In Fig. 2, for instance, intervals a_1, a_2, a_3, and b correspond to the balances of two bank accounts a and b (the key axis denotes the balance amount), both of which are created at time t_0 and cancelled at t_3. For account a, there is one withdrawal at t_1 and one deposit at t_2, while b remains constant (10k) during the period $[t_0, t_3)$. In the

2. The fractal dimensionality of a data set differs from the "embedding dimensionality" (i.e., the number of axes in the data space) in that, it describes the "intrinsic" characteristics of the distribution. For example, for a set of 2D points on a line, its fractal dimensionality equals 1 (while its embedding dimensionality is 2). We refer our readers to [14] for a more detailed introduction to fractal concepts.

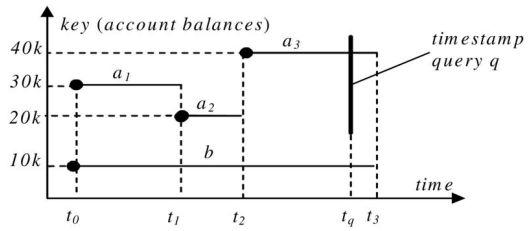

Fig. 2. Representation of temporal data.

sequel, we represent an interval as *lifespan:key*, where the *lifespan* indicates the validity period of the *key*. For example, the lifespan of a_1 is $[t_0, t_1)$ (excluding t_1) and a_1 is represented as $[t_0, t_1)$:30k. A *timestamp range query* q_t: $[q_{k1}, q_{k2}]$ retrieves all records whose key values are in the range $[q_{k1}, q_{k2}]$ at time q_t. The query in Fig. 2 ("find accounts whose balances at time q_t are between 15k and 45k dollars") outputs the horizontal line segments (a_3) intersecting the vertical line segment q.

The MVB-tree [7] utilizes the multiversion framework [13] for optimal processing of timestamp range queries in external memory. The tree is constructed in chronological order of the interval lifespans. Specifically, an interval is inserted at its starting time; in Fig. 2, for example, a_1 and b are inserted at time t_0 as $[t_0, ^*)$:30k and $[t_0, ^*)$:10k, respectively, where "*" means that their lifespans progress with time (i.e., the records are *alive*). On the other hand, an interval is *logically deleted* at its ending time by terminating its lifespan (i.e., the record *dies*, but is not physically removed from the database). At t_1, for instance, interval a_1 dies and its lifespan is modified to $[t_0, t_1)$.

Fig. 3 illustrates a MVB-tree (at time 3), where each entry has the form $< key, t_{start}, t_{end}, pointer >$. For leaf entries, the *pointer* (omitted in the figure for clarity) points to the actual record with the corresponding *key* value, while for intermediate entries, it points to a node at the next level. The temporal attributes t_{start} and t_{end} denote the time that the record was inserted and (logically) deleted, respectively. For each timestamp t and every node except the roots, it is required that either none, or at least $B \cdot P_{version}$ entries are alive at t, where $P_{version}$ is a tree parameter and B the node capacity (for Fig. 3 and the following examples, $P_{version} = 1/3$ and $B = 6$). This *weak version condition* ensures that entries alive at the same timestamp are mostly grouped together in order to optimize timestamp queries. Violations of this condition generate *weak version underflows*, which occur as a result of (logical) deletions.

Insertions and deletions are carried out in a way similar to B-trees except that overflows and underflows are handled differently. A *block overflow* occurs when an entry is inserted into a full node (already containing B records), in which case a *version split* is performed. To be specific, all the live entries of the node are copied to a new node, with their t_{start} modified to the insertion time. The t_{end} of these entries in the original node is changed from * to the insertion time.[3] In Fig. 4, the insertion of $< 28, [4, ^*) >$ at timestamp 4 (in the tree of Fig. 3) causes node A to overflow. A new node D is

3. In practice, this step can be avoided since the deletion time is implied by the entry in the parent node.

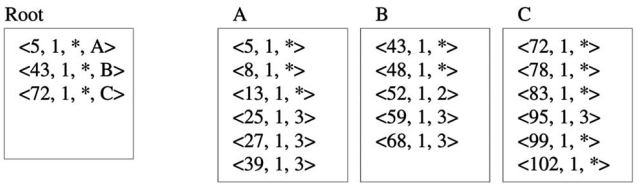

Fig. 3. A MVB-tree example.

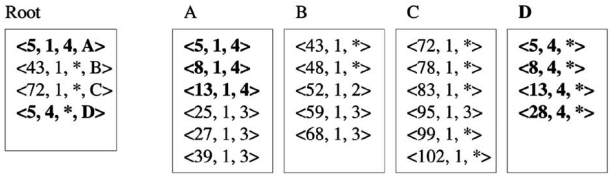

Fig. 4. Example of block overflow and version split.

created to store the live entries of A, and A dies (notice that all * are replaced by 4) meaning that it will not be modified in the future (bold entries indicate the changes from Fig. 3).

In some cases, the new node may be almost full so that a small number of insertions would cause it to overflow again. On the other hand, if it contains too few entries, a small number of deletions will cause it to underflow. To avoid these problems, it is required that the number of entries in the new node must be in the range $[B \cdot P_{svu}, B \cdot P_{svo}]$, where P_{svo} and P_{svu} are tree parameters (for the following examples, $P_{svu} = 1/3$, $P_{svo} = 5/6$). A *strong version overflow* (*underflow*) occurs when the number of entries exceeds $B \cdot P_{svo}$ (becomes lower than $B \cdot P_{svu}$). A strong version overflow is handled by a *key split*, a version-independent split according to the key values of the entries in the block. Notice that the strong version condition is only checked after a version split, i.e., it is possible that the live entries of a node are above $b \cdot P_{svo}$ due to subsequent insertions.

Strong version underflows are similar to weak version underflows, the only difference being that the former happen after a version split, while the latter occur when the weak version condition is violated. In both cases, a merge is attempted with the copy of a sibling node using only its live entries. If the merged node strong version overflows, a key split is performed. Assume that at timestamp 4 we want to delete entry $< 48, [1, ^*) >$ from the tree in Fig. 3. Node B weak version-underflows since it contains only one live entry $< 43, [1, ^*) >$. A sibling, let node C, is chosen and its live entries are copied to a new node C'. The inclusion of $< 43, [4, ^*) >$ into C' causes strong version overflow, leading to a key split and finally nodes D and E are created (Fig. 5).

The MVB-tree essentially includes, in a space efficient manner, a (logical) B-tree at each timestamp consisting of all entries whose lifespans cover the corresponding time. Processing a timestamp query retrieves the root of the B-tree at the query timestamp, after which the search proceeds like a normal B-tree, guided by key, t_{start}, and t_{end}. As shown in [7], the MVB-tree requires $O(N/B)$ space, where N is the number of data intervals and B the node capacity, and answers a timestamp range query in $O(\log_B M + K/B)$ node accesses, where M is the number of live intervals at the queried timestamp and K is the

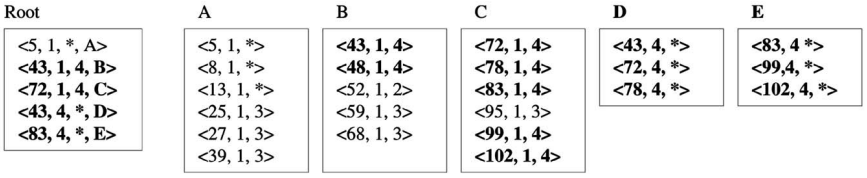

Fig. 5. Example of weak version underflow.

number of output intervals. Both the space consumption and query performance are asymptotically optimal. A variation of MVB-trees, which reduces the tree size by a constant factor can be found in [36]. Notice that, the construction algorithms of the MVB-tree (as well as the related underflow/overflow concepts) are not specific to B-trees, but can be applied to other conventional indexes (e.g., R-trees) to obtain the corresponding multiversion structures [25], [33], [24]. The performance (including space requirements and query cost) of general multiversion structures is studied in [34].

3 Asymptotical Performance Analysis of the Aggregate R-Tree

The analysis of Section 2.1 focuses on the practical query cost of the aR-tree (i.e., in terms of node accesses), leaving its asymptotical performance open. It can been shown [1] that in the worst case (given some specially-designed data sets) a RA query may access all nodes in the aR-tree, resulting in $\mathrm{O}(N/B)$ query cost. Nevertheless, aR-trees (in general, all the R-tree-family structures) are known to perform well for practical data sets and have been applied extensively in the literature. Thus, judging their quality by its worst-case cost can be misleading. An important and useful question should be: What is their asymptotical performance in the *expected case*?

In the sequel, we answer this question by utilizing the some of the results presented in Section 2.1. Specifically, our goal is to rewrite (6) as a function of the data set cardinality. For this purpose, it suffices to focus on 1) the number of leaf nodes visited, which asymptotically dominates the overall query cost and 2) queries whose sizes are larger than the leaf MBBs (so that in (4), we can ignore the second case) since their costs absorb those of smaller queries. To facilitate the derivation, we consider queries that have the same length (denoted as q_s) on all axes (the results hold for general queries as well).

3.1 Uniform Data Distribution

The solution for the uniform case is relatively easy by expanding (4) (i.e., the access probability $PR_{acs-0}(q, s_0)$ of a leaf node) as follows (by the Binomial Theorem):

$$\begin{aligned} PR_{acs-0}(q, s_0) &= (q_s + s_0)^d - (q_s - s_0)^d \\ &= \sum_{i=0}^{d} \left[\binom{d}{i} q_s^i s_0^{d-i} \right] - \sum_{i=0}^{d} \left[\binom{d}{i} q_s^i (-s_0)^{d-i} \right]. \end{aligned} \tag{7}$$

We relate the cost to the number K of data points in the query hyper-rectangle. Since the distribution is uniform, K equals $N \cdot q_s^d$ (where N is the data set cardinality and q_s^d the volume of the query hyper-rectangle), leading to $q_s = (K/N)^{1/d}$. Further, according to (5), $s_0 = (f/N)^{1/d}$, where f is the average node fanout. Thus, (7) can be rewritten as:

$$\begin{aligned} &PR_{acs-0}(q, s_0) \\ &= \sum_{i=0}^{d} \left[\binom{d}{i} \left(\frac{K}{N}\right)^{\frac{i}{d}} \left(\frac{f}{N}\right)^{\frac{d-i}{d}} \right] - \sum_{i=0}^{d} \left[\binom{d}{i} \left(\frac{K}{N}\right)^{\frac{i}{d}} \left(\frac{-f}{N}\right)^{\frac{d-i}{d}} \right] \\ &= \frac{1}{N} \left\{ \sum_{i=0}^{d} \left[\binom{d}{i} K^{\frac{i}{d}} f^{\frac{d-i}{d}} \right] - \sum_{i=0}^{d} \left[\binom{d}{i} K^{\frac{i}{d}} (-f)^{\frac{d-i}{d}} \right] \right\}. \end{aligned} \tag{8}$$

Since the cardinality N and the fanout f are constants, (8) is asymptotically dominated by the highest power of K that is not canceled between the first and second summations, namely, $K^{(d-1)/d}$ obtained at $i = d-1$ (note that the term with $i = d$ is canceled). Hence, we have: $PR_{acs-0}(q, s_0) = \mathrm{O}(\frac{1}{N} K^{(d-1)/d} B^{1/d})$. Given that there are totally N/f leaf nodes, the number $Cost_0(q)$ of leaf accesses is

$$\frac{N}{f} \cdot PR_{acs-0}(q, s_0) = \frac{N}{f} \mathrm{O}\left(\frac{1}{N} K^{(d-1)/d} B^{1/d} \right) = \mathrm{O}((K/B)^{(d-1)/d})$$

(applying $f = \mathrm{O}(B)$).

This result leads to several interesting observations. First, although a RA query does not retrieve the points inside the query hyper-rectangle, the number K of such points actually determines the query cost. Particularly, the cost monotonically increases with the query size (larger queries incur higher K), eventually reaching $\mathrm{O}((N/B)^{(d-1/d)})$, which can be prohibitive in practice. This is disappointing because the aR-tree was originally motivated by the need to efficiently process large queries (recall that its performance is similar to that of the normal R-tree for small queries). Second, the benefit of the aR-tree decreases as the dimensionality increases; for high d, the complexity approaches that of a simple sequential scan when $K = \mathrm{O}(N)$ (i.e., large queries).

3.2 Arbitrary Data Distribution

A similar result holds for any data distribution, except that d in $\mathrm{O}((N/B)^{(d-1/d)})$ will become d', i.e., the data set's fractal dimensionality. To prove this, as before, we need to rewrite (6) as a function of the data set cardinality, which, however, is more complex because, since d' is not an integer, an expansion similar to that of (7) no longer holds.[4]

4. Although the Binomial Theorem also applies to real exponents (leading to a series), the application of the theorem here would require a lengthy discussion on the convergence of the resulting series.

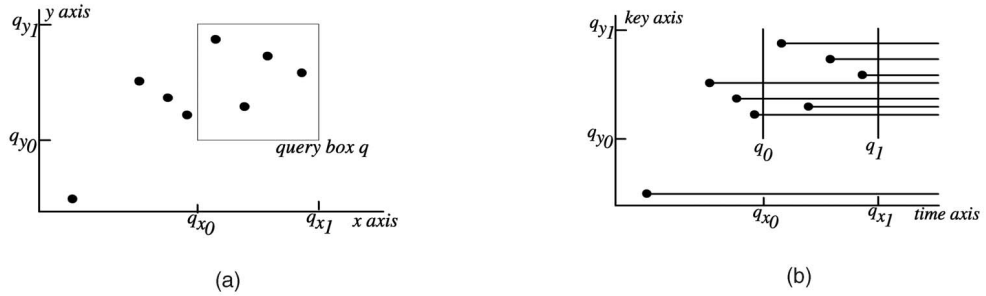

Fig. 6. Reduction of range aggregate to vertical range aggregate. (a) Original points and query. (b) Converted intervals and queries.

In the sequel, we provide a proof utilizing the Maclaurin expansion.

We start with a few preliminaries. First, it is well known that the fractal dimension d' must be in the range $[1, d]$ [14]. Second, a query with extent q_s on each dimension retrieves on average $K = N \cdot q_s^{d'}$ points [6], so that $q_s = (K/N)^{1/d'}$. Assuming large data sets (i.e., high N) and large queries (particularly, the number K of points in q should be at least B), then q_s is longer than the leaf node extent $s_0 = (f/N)^{1/d'}$ (as in (5)). The goal is to establish $\frac{N}{f} \cdot PR_{acs-0}(q, s_0) = \mathrm{O}(K/B)^{(d'-1)/d'}$. Towards this, we have:

$$\begin{aligned}
&\frac{N}{f} PR_{acs-0}(q, s_0) = (N/f)\left[(q_s + s_0)^d - (q_s - s_0)^d\right] \\
&= (N/f)\left\{\left[(K/N)^{1/d'} + (f/N)^{1/d'}\right]^{d'} - \left[(K/N)^{1/d'} - (f/N)^{1/d'}\right]^{d'}\right\} \\
&= \left[(K/f)^{1/d'} + 1\right]^{d'} - \left[(K/f)^{1/d'} - 1\right]^{d'} \\
&= \mathrm{O}\left\{\left[(K/B)^{1/d'} + 1\right]^{d'} - \left[(K/B)^{1/d'} - 1\right]^{d'}\right\}.
\end{aligned} \tag{9}$$

Our objective now is to prove

$$[(K/B)^{1/d'} + 1]^{d'} - [(K/B)^{1/d'} - 1]^{d'} = \mathrm{O}(K/B)^{(d'-1)/d'}.$$

Dividing both sides with K/B, this equation is equivalent to

$$[1 + (B/K)^{1/d'}]^{d'} - [1 - (B/K)^{1/d'}]^{d'} = \mathrm{O}(B/K)^{1/d'}.$$

Let $t = (B/K)^{1/d'}$; then, the target is to prove $(1+t)^{d'} - (1-t)^d = \mathrm{O}(t)$ as $t \to 0$, i.e., K approaches N (which is a large number). Further, recall that $K \geq B$, so $0 \leq t \leq 1$. By the Maclaurin expansion,

$$(1+t)^{d'} = 1 + d' \cdot t + [d' \cdot (d'-1)/2 \cdot (1+t^*)^{d'-2}] \cdot t^2,$$

and

$$(1-t)^d = 1 - d' \cdot t + [d' \cdot (d'-1)/2 \cdot (1-t^{**})^{d'-2}] \cdot t^2,$$

where t^* and t^{**} are some values in $[0, t]$. Notice that, for t^* and t^{**} in $[0, 1]$,

$$(1+t^*)^{d'-2} - (1-t^{**})^{d'-2} = \mathrm{O}(\max\{1, 2^{d'-2}\}) = \mathrm{O}(1).$$

Therefore, $(1+t)^{d'} - (1-t)^d = 2 \cdot d' \cdot t + \mathrm{O}(t^2) = \mathrm{O}(t)$ (since $\mathrm{O}(t^2) = \mathrm{O}(t)$ for $t \to 0$); thus, completing the proof. We close this section with the following theorem.

Theorem 1. *Given a set of N points whose fractal dimensionality is d', the average query performance of the aR-tree is $O(N/B)^{(d'-1)/d'}$, where B is the disk page capacity.*

Motivated by the large query cost of the aR-tree, in the next section, we propose the aP-tree, which achieves logarithmic cost at the expense of higher space consumption. Compared to other existing structures for RA-queries, such as the MVSB-tree and the CRB-tree, the aP-tree is simpler to implement since it requires only marginal changes to the popular multiversion framework and does not make stringent assumptions about the memory storage.

4 The Aggregate Point-Tree (aP-Tree)

Starting with two-dimensional data, Sections 4.1 and 4.2 describe the basic idea behind the aP-tree, and its concrete (construction and query) algorithms, respectively. Section 4.3 provides a detailed analysis on the (asymptotical and practical) performance of the new structure, and Section 4.4 focuses on techniques making the aP-tree dynamic. Finally, Section 4.5 generalizes the solution to higher dimensionalities.

4.1 Overview

Let us consider the 2D universe, where the range of the x-(y-)dimension is $[0, X]$ ($[0, Y]$). We convert each data point (p_x, p_y) to a horizontal interval $[p_x, X]{:}p_y$ (adopting the interval representation of Section 2.3) with x-(y-)projection $[p_x, X]$ (p_y). Fig. 6a shows the points used in Fig. 1a and Fig. 6b illustrates the transformed intervals. If we consider the x-(y-)axis as the time (key) dimension, each interval $[p_x, X]{:}p_y$ in Fig. 6b can be regarded as a record in the key-time space with key p_y and lifespan $[p_x, X]$. Particularly, the lifespans of all intervals terminate at the maximum time-stamp X (i.e., corresponding to "*" in Section 2.2). In the sequel, we use the x-y and key-time notations interchangeably, whichever is more convenient.

We represent a 2D RA query rectangle q as $[q_{x_0}, q_{x_1}]{:}[q_{y_0}, q_{y_1}]$, denoting its ranges on both dimensions. As shown in Fig. 6b, the number of points in q, equals the number of intervals that intersect the vertical line segment q_1 (i.e., the right boundary of q represented as $q_{x_1}{:}[q_{y_0}, q_{y_1}]$), but not q_0 (i.e., the left boundary represented[5] as $\rightarrow q_{x_0}{:}[q_{y_0}, q_{y_1}]$). Thus, a RA query

5. The symbol $\rightarrow$ indicates that the x-coordinate of q_0 infinitely approaches (but does not equal) x_0 from the left. This distinction is necessary to include points on the left boundary of the query rectangle q.

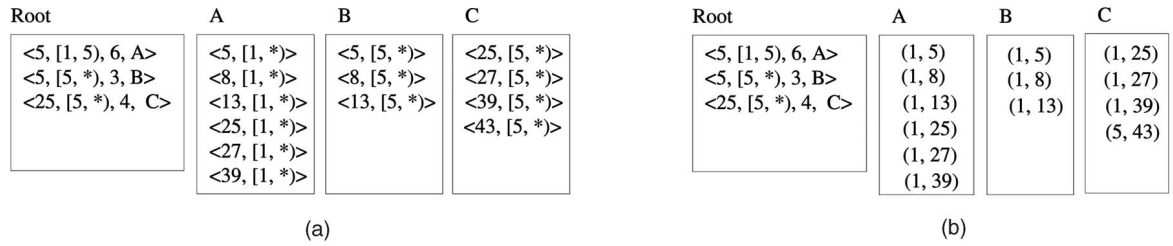

Fig. 7. An aP-tree. (a) Example. (b) Alternative representation.

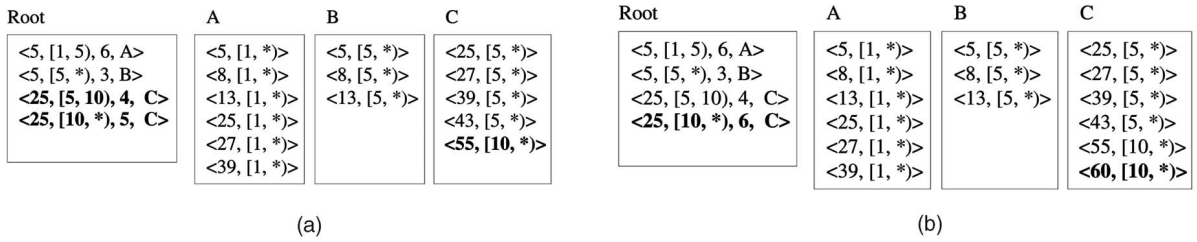

Fig. 8. Insertions that do not trigger overflows. (a) Duplication of an intermediate entry is necessary. (b) Duplication is not necessary.

is reduced to two *vertical range aggregate* (VRA) queries q_1 and q_0. Specifically, in the key-time space, $q_1 = q_{x_1}:[q_{y_0}, q_{y_1}]$ retrieves the number q_{N_1} of intervals that start before or at timestamp q_{x_1}, and their keys are in the range $[q_{y_0}, q_{y_1}]$. Similarly, $q_0 = \rightarrow q_{x_0}:[q_{y_0}, q_{y_1}]$ returns the number q_{N_2} of intervals that start before (but not at) timestamp x_0. The result of the original RA query equals $q_{N_2} - q_{N_1}$.

A VRA query resembles the timestamp range query, optimally solved by the MVB-tree, except that here we are interested only in the aggregate number, instead of the concrete points (or intervals in the key-time space). This fact differentiates query processing and, therefore, affects the corresponding index structure. In the next section, we show how the aP-tree avoids the retrieval of the intervals intersecting q_1 and q_0, as well as, the expensive computation of their set difference.

4.2 Algorithms of the aP-Tree

Similar to the multiversion B-tree, the entry format of the aP-tree is $< y, [x_{start}, x_{end}), agg, pointer >$, where

$$y, [x_{start}, x_{end}), pointer$$

correspond to the fields $key, [t_{start}, t_{end}), pointer$ in the MVB-tree, respectively. For a leaf entry (representing a data point), the additional field *agg* equals 1 for the *count* aggregate;[6] for an intermediate entry, it denotes the number of leaf entries in its subtree alive in $[x_{start}, x_{end})$. In the sequel, we refer to the fields y and $[x_{start}, x_{end})$ of each entry as its *key* and *lifespan*, respectively. Fig. 7a illustrates a simple example (omitting *agg* and *pointer* of leaf entries) for seven points at coordinates (1,5), (1,8), (1,13), (1,25), (1,27), (1,39), (5,43), assuming node capacity of six. The leaf entry $< 5, [1,^*) >$ in node A, for example, refers to the horizontal interval $[1, X]:5$ transformed from point (1,5). The intermediate entry $< 5, [1, 5), 6, A >$ implies that there are six entries in node A, which are alive in interval $[1, 5)$ and whose keys are at least five. Fig. 7b shows the equivalent tree where the leaf entries are represented in (p_x, p_y) format. Notice that some points (e.g., (1,5), (1,25)) are replicated in multiple nodes by the tree construction algorithms discussed below.

4.2.1 Tree Construction

As with MVB-trees, the transformed intervals are inserted into an aP-tree in ascending order of their starting timestamps, i.e., the x-coordinates of the original data points (an external sorting is necessary to achieve this order). An important difference is that no (logical) deletion is necessary in aP-trees and the lifespans of all the leaf intervals terminate at the maximum timestamp X. Consequently, the number of live entries in any node keeps increasing until it dies, and weak/strong version underflows never happen. Thus, the aP-tree has only a single parameter P_{svo} (no parameters $P_{version}$ and P_{svu}), which denotes the strong version overflow threshold.

Insertion is similar to that in MVB-trees except that it may be necessary to duplicate intermediate entries on the path. As an example, assume that an interval $< 55, [10,^*) >$ (equivalently, point (10,55)) is inserted into the tree in Fig. 7. First, the leaf node (i.e., C) that accommodates the new entry is identified (following the intermediate entry $< 25, [5,^*), 4, C >$). Then, as shown in Fig. 8a, the following changes are applied to the root node: 1) A new entry is duplicated from $< 25, [5,^*), 4, C >$ with, however, its x_{start} set to 10, and its *agg* incremented by one (to 5) and 2) entry $< 25, [5,^*), 4, C >$ dies, having its x_{end} modified to 10. Such entry duplication is necessary for the aP-tree (but not for the MVB-tree) to ensure that the *agg* field correctly reflects the aggregate result during the entry's lifespan $[x_{start}, x_{end})$. In general, duplication is required when the x_{start} of the intermediate entry is smaller than that of the interval being inserted. Fig. 8b shows the aP-tree after the insertion of interval $< 60, [10,^*) >$, where no new intermediate entry is spawned (but the *agg* of the parent entry of C is incremented to six).

6. For *count*, the *agg* field can be omitted for leaf nodes. In case of the *sum* aggregate function, *agg* stores the weight of the point (leaf nodes), or the sum of the weights in the corresponding subtree (for intermediate nodes).

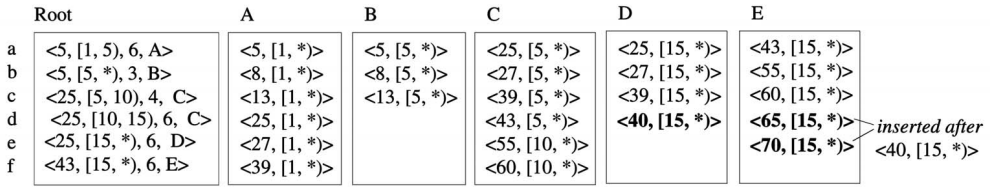

Fig. 9. Overflow of leaf node C.

Algorithm Insert (p_x, p_y, *nd*)
/* (p_x, p_y) is the coordinate of the point being inserted; *nd* is the (leaf/non-leaf) node where the insertion is being made */
1. if *nd* is a leaf node
2. insert entry p_y:[p_x,*) into *nd*
3. if *nd* overflows
4. version copy *nd* into another node, and split it into new nodes nd_1 and nd_2
5. else
6. find the live entry e_y:[e_x,*) whose e_y is the largest among all the entries with $e_x \leq p_x$
7. **Insert**(p_x, p_y, $e_{childnode}$) //insert (p_x, p_y) to the next level
8. if $e_x < p_x$ //returning from the next level
9. set the lifespan of e to [e_x, p_x)
10. create a new entry e' s.t. $e'_y = e_y$, $e'_{lifespan} = [p_x, *)$, and $e'_{agg} = e_{agg} + 1$
11. else e_{agg}++
12. if the child node splits set/create entries pointing to the new node(s) appropriately
13. if *nd* overflows
14. version copy *nd* into another node nd_1
15. if nd_1 strong version overflow then split it into itself and nd_2
End Insert

Fig. 10. The aP-tree insertion algorithm.

An overflow of a leaf node always triggers a strong version overflow and is handled in the same way as the MVB-tree. Specifically, all the entries in the overflowing node are *version copied* to a new node, which is then split into two using a key split. Fig. 9 illustrates an example, where $< 40, [15,^*) >$ is inserted into the full node C in the tree of Fig. 8b, causing it to overflow. All the entries in C are copied to a new node (with their t_{start} set to 15), which splits into node D and E. The parent entry of C dies and two entries are added into the root node pointing to D and E, respectively. Due to the absence of logical deletions, the x_{end} fields of leaf entries are never modified (i.e., they always remain "*"), which means that they do not need to be actually stored. Fig. 9 shows the final aP-tree after inserting intervals $< 65, [15,^*) >$ and $< 70, [15,^*) >$ (as with Fig. 8, no intermediate duplication is necessary).

An overflow of an intermediate node is processed in the same way as the MVB-tree; specifically, a new node is created through a version split, after which a key split is performed if the number of leaf entries exceeds $B \cdot P_{svo}$, where B is the node capacity and P_{svo} is the strong version overflow threshold. The complete insertion algorithm is summarized in Fig. 10.

4.2.2 Bulkloading the aP-Tree

Motivated by the *buffer-tree* technique, proposed in [2] and later refined in [9], we present a bulkloading algorithm for the aP-tree by associating each node with a constant number of *pooling* disk pages (for the following examples, we assume the number is 1 and each pooling page can contain up to six entries). The idea is to "stall" insertions in the pooling pages (of various parts of the tree) and, then, process entries of a full pooling page in a batched manner. We illustrate this by bulkloading the data points (i.e., intervals) of Fig. 9 (recall that the entries of the tree in Fig. 9 correspond to 12 points). The first six intervals (i.e., in node A) are inserted into the (initially empty) root. Since the node capacity is six, incremental insertion of the next interval $< 43, [5,^*) >$ would incur an overflow, leading to a strong version overflow and a tree with two levels, thus doubling the costs of subsequent insertions. Instead, the bulkloading algorithm simply places the new interval in the pooling page of the root (i.e., stalling the insertion process).

Fig. 11a shows the tree after stalling the subsequent five intervals in the same way. Note that each "stalling" requires accessing only one (pooling) page, in contrast to the tree-height cost of the incremental algorithm. The insertion processes in a pooling page are "resumed" when the page gets full, as is the case in Fig. 11a. The first pooled entry $< 43, [5,^*) >$ triggers a strong version overflow in the root. This is handled in the same way as incremental insertion: creating a new root and two leaves B, C (resulting in the same structure as in Fig. 7a, where node A corresponds to the root in Fig. 11a). Then, the algorithm decides, for each remaining entry in the pool, which of the two leaves will accommodate it (in this case node C for all entries). C is loaded *only once* and accommodates $< 55, [10,^*) >$ and $< 60, [10,^*) >$ before it gets full. Thus, the other three entries (from the pool of the root) are stalled again, but this time in the pooling page of C, as in Fig. 8b. The pool of the root is now empty; if there were more records, they would be simply stalled there.

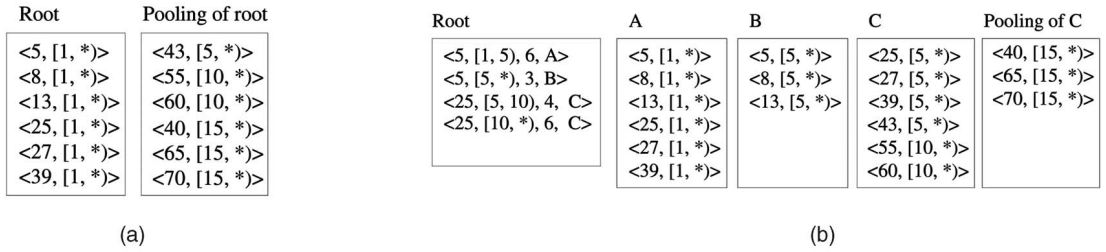

Fig. 11. Bulkloading the aP-tree using pooling pages. (a) After inserting 12 entries. (b) After processing the entries pooled in root.

In our example, there is no more data and, thus, the insertions pooled at C are resumed. As before, the first entry $< 40, [15,^*) >$ in the pool triggers a structural change (strong version overflow of C) which is handled as in Fig. 9, creating leaf nodes D and E. Then, the algorithm decides the leaf nodes that will include the other two pooled entries; in this example, both of them should be inserted to E, which is loaded once to complete the whole construction. The final tree has exactly the same structure as in Fig. 9. Compared to incremental insertion, the above bulkloading algorithm retrieves the nodes affected by multiple insertions only once, leading to smaller amortized overhead.

4.2.3 VRA Query Algorithm

As discussed in Section 4.1, the aP-tree answers a RA query $[q_{x_0}, q_{x_1}]:[q_{y_0}, q_{y_1}]$ using two VRA queries $q_1 = q_{x_1}:[q_{y_0}, q_{y_1}]$ and $q_0 =\rightarrow q_{x_0}:[q_{y_0}, q_{y_1}]$. The processing of q_1 starts by locating the corresponding aggregate B-tree for q_{x_1} (recall that an aP-tree includes a logical aggregate B-tree at every x-coordinate), after which the search is guided by the keys and lifespans of intermediate entries. For example, in Fig. 9, to answer the VRA query 15:[25,45], we only consider the root entries whose lifespans include 15; thus, entries a, c, d are eliminated immediately. Among the remaining entries, we purge b because the keys in its subtree are in the range [5,25) (25 is inferred from the key value of e), which does not intersect the query y-range [25,45]. For entry e, whose key-range [25,43] is covered by [25,45], it suffices to simply add its *agg* value (i.e., 6) without accessing its subtree. We only need to descend those entries (e.g., f) whose key-ranges [43, ∞] partially intersects [25,45]. In this case, the algorithm only visits one leaf node D where all four intervals intersect the query, leading to the final answer $6 + 4 = 10$. The processing of VRA query $q_0 =\rightarrow q_{x_0}:[q_{y_0}, q_{y_1}]$ is the same, except that intervals starting at q_{x_0} should be excluded. As an example, consider the query $\rightarrow 15:[25, 45]$. After retrieving the corresponding root node, we eliminate entries (a, c, e, f) that start at or after 15. Then, the processing proceeds as in the previous case by examining the key-ranges of the entries. Finally, the result 4 is returned by visiting leaf node C.

An important observation is that, in the worst case, a VRA query can be answered by visiting two paths from the root to the leaf level of a logical B-tree. This is because, at each intermediate level, the key-ranges of the entries alive at any x-coordinate (e.g., $\mathrm{x} = 15$ in the query example above) are continuous and disjoint. Consequently, there can be at most two key-ranges partially intersecting the y-range of the query (enclosing the starting, or ending point of the range). The key-ranges of the other entries are either contained by (in which case their *agg* fields are simply added) or disjoint with the query y-range (in which case their subtrees are not visited). Therefore, *the number of node accesses of a RA query is at most four times the height of the B-tree* (two times for each q_1 and q_0).

4.3 Performance Analysis

We divide our analysis into two parts, focusing on the asymptotical and practical performance, respectively. Our derivation uses the symbols listed in Table 1.

4.3.1 Asymptotical Performance

We first analyze the asymptotical space consumption of the aP-tree. Recall that a node is created from a version split and dies by generating another version split (which spawns a new node). When a node is created, it is guaranteed to contain fewer than $P_{svo} \cdot B$ entries (where P_{svo} is the threshold for strong version overflow and B the node capacity); otherwise, a strong version overflow occurs and a key split is performed. It follows that a node will generate the next version split by receiving at least $(B - P_{svo} \cdot B)$ insertions after its creation. Let N_0 be the number of leaf nodes. Since each insertion will create only one entry in a leaf node, we have:

$$N_0 \cdot (B - P_{svo} \cdot B) \leq N \Rightarrow N_0 \leq \frac{N}{(1 - P_{SVO})B} = \mathrm{O}\left(\frac{N}{B}\right).$$

For a node at higher levels, the number of new entries created by an insertion is at most 2, corresponding to the two parent entries for the new nodes created at the next level (through strong version overflows). Hence, an intermediate node dies after at least $(B - P_{svo} \cdot B)/2$ insertions. Assuming that the number of nodes at level i is N_i, we have:

$$N_i \cdot (B - P_{svo} \cdot B)/2 \leq N \Rightarrow N_i \leq \frac{2N}{(1 - P_{SVO})B} = \mathrm{O}\left(\frac{N}{B}\right).$$

Since every node contains at least $P_{svo} \cdot B/2 = \mathrm{O}(B)$ entries at each point during its lifespan, the height of any B-tree is $\mathrm{O}(\log_B(N/B))$. Therefore, the space complexity[7] of the aP-tree is $\mathrm{O}((N/B)\log_B(N/B))$ nodes in the worst case. Furthermore, since, as with conventional B-trees, each insertion incurs at most $2\log_B N$ node accesses, the aP-tree can be incrementally constructed with $\mathrm{O}(N \log_B(N/B))$ node

7. The aP-tree consumes more space than the MVB-trees $\mathrm{O}(N/B)$ due to entry duplication that occurs in the aP-tree when the x_{start} of an intermediate entry is smaller than the x_{start} of the interval being inserted, as shown in Fig. 8.

TABLE 1
Frequently Used Symbols

Symbol	Meaning
P_{svo}	Strong version overflow threshold of the aP-tree
B	Disk page size
N	Dataset cardinality
h	Height of the tree
f_l (f_{nl})	Live fanout of a leaf (non-leaf) node
SP_l (SP_{nl})	Split point of a leaf (non-leaf) node
B_l (B_{nl})	Node capacity of a leaf (non-leaf) node
L_i	Level up point of a node at the i-th level
N_i	Number of nodes at the i-th level of the tree

accesses, which also dominates the cost of the preprocessing sorting step. Applying the analysis in [2], [9], we can show that the aP-tree can be bulkloaded in $O((N/B)\log_B(N/B))$ using the buffer-tree technique of Section 4.2. Answering a VRA query involves visiting at most two paths from the root of an aggregate B-tree to the leaf level, i.e., $2\log_B N = O(\log_B(N/B))$ node accesses in the worst case. Given that, a WA query is transformed into two VRA queries, a RA query can also be answered in $O(\log_B(N/B))$ node accesses in the worst case.

Theorem 2. *A RA query for two-dimensional points can be answered in $O(\log_B(N/B))$ query time by building an aP-tree consuming $O((N/B)\log_B(N/B))$ space that can be incrementally constructed in $O(N\log_B(N/B))$ node accesses, or bulkloaded in $O((N/B)\log_B(N/B))$ time.*

4.3.2 Cost Models for aP-Trees

Next, we analyze the practical performance of the aP-tree, aiming at quantifying the tradeoff between its size consumption and query response time. It is worth mentioning that the performance analysis of multiversion structures in [34] is not applicable here because it assumes an equal number of insertions and (logical) deletions (while the aP-tree does not involve deletions). We start by estimating the size of the aP-tree, considering the general case where all points have different x-coordinates. Let the *live fanout* f_l of a leaf node be the average number of entries in the node alive at an x-coordinate during the node's lifespan. Similarly, f_{nl} represents the *live fanout* of an intermediate node. For example, in node C of Fig. 9, there are four entries alive at $x = 5$ and six at $x = 10$. So, the live fanouts of C are 3 and 4, respectively, at these two x-coordinates. Note that leaf and intermediate nodes are distinguished because their entry formats are different.

The number of live nodes at some x-coordinate increases due to key splits. Recall that when an overflow occurs at the leaf level, the new leaf node will always be key split, while, for intermediate levels, key splits happen only when the number of entries in the new node exceeds the strong version overflow threshold $B \cdot P_{svo}$. To distinguish this, we define the *split point* SP_l of a leaf node as the number of entries it contains when being key split. Similarly SP_{nl} corresponds to the *split point* of an intermediate node. If B_l and B_{nl} are the block capacities of leaf and intermediate nodes, respectively, we have:

$$SP_l = B_l \text{ and } SP_{nl} = B_{nl} \cdot P_{svo}. \tag{10}$$

As shown in [38], the fanout of a B-tree is $ln2$ times the split point of a node. Hence, in our case, the relation between live fanouts and split points is:

$$f_l = SP_l \cdot ln2 \text{ and } f_{nl} = SP_{nl} \cdot ln2. \tag{11}$$

An aP-tree consists of multiple logical (aggregate) B-trees; trees corresponding to larger x-coordinates index more intervals and hence have higher heights. The height h of the logical B-tree at the largest x-coordinate is:

$$h = 2 + \left\lceil \log_{f_{nl}} \frac{N/f_l}{SP_{nl}} \right\rceil. \tag{12}$$

If N_i is the total number of nodes at level i, the size of an aP-tree is:

$$Size_{aP} = \sum_{i=0}^{h-1} N_i. \tag{13}$$

The estimation for N_0, the total number of leaf nodes, is relatively easy, observing that the only type of structural change at the leaf level is a version split followed by a key split. Therefore, each version split 1) increases the total number of nodes by two and 2) the number of live nodes by one. Notice that, after all the insertions are complete, the number of live nodes is N/f_l; thus, the total number of leaf-level version splits is $V_l = n/f_l - 1$. Hence, we have:

$$N_0 = 2V_l + 1 = 2N/f_l - 1. \tag{14}$$

A similar analysis, however, does not apply to the estimation for N_i of intermediate levels because key splits do not always happen after version splits. Furthermore, note that higher levels will appear only after a sufficient number of insertions. In the sequel, we say that the *level-up point* (LuP) for level i is L_i, if this level appears after L_i insertions. Since a new level appears when the previous root at the lower level strong version overflows, the estimation for L_i ($i \geq 1$) is as follows:

$$L_1 = SP_l \text{ and } L_i = f_l \cdot f_{nl}^{i-2} \cdot SP_{nl} \quad (1i \leq h-1), \tag{15}$$

where SP_l, SP_{nl} and f_l, f_{nl} are split points and live fanouts for leaf and intermediate nodes, respectively. Next, we focus on N_1 before generalizing to higher levels. Since no two points have the same x-coordinate, an entry will be duplicated in every intermediate node along the insertion

path. Therefore, the total number of insertions at each level is also N. This estimation excludes the inserted entries due to strong version overflows, which is not a problem because the number of strong version overflows is considerably lower than N; thus, omitting them will not bias the results significantly.

Recall that a node already contains a number of entries (version copied from the previous node) when it is created. Further, this number equals the number of live entries in the previous node. Since the average live fanout of intermediate nodes is f_{nl}, it follows that a node contains f_{nl} initial entries. Therefore, a node will, on the average, take $(B_{nl} - f_{nl})$ entries before it dies. However, the live fanout applies only to nodes other than roots of logical trees (i.e., for N_1, it applies after level 2 has appeared). Hence, the number of level 1 nodes, created after the LuP L_2, can be estimated as $(N - L_2)/(B_{nl} - f_{nl})$, where L_2 is given in (15).

At any time between LuPs L_1 and L_2, there is only one live node at level 1, which is the root of the logical tree. The live entries in the root increase gradually from 2 (when level 1 appears) to SP_{nl} (when level 2 appears). It follows that on the average, $(L_2 - L_1)/(SP_{nl} - 2)$ insertions are performed before the live entries in the root increase. For each value of j, by the same analysis as above, the number of newly created nodes is

$$\frac{(L_2 - L_1)/(SP_{nl} - 2)}{(B_{nl} - j)};$$

thus, we have the following estimation for N_l:

$$N_1 = \left[\sum_{j=2}^{SP_{nl}} \frac{(L_2 - L_1)/(SP_{nl} - 2)}{B_{nl} - j}\right] + \frac{N - L_2}{B_{nl} - f_{nl}}. \quad (16)$$

Similar analysis also applies to higher levels, except level $h - 1$. In general, we have:

$$N_i = \left[\sum_{j=2}^{SP_{nl}} \frac{(L_{i+1} - L_i)/(SP_{nl} - 2)}{B_{nl} - j}\right] + \frac{N - L_{i+1}}{B_{nl} - f_{nl}} \quad (1 \le i \le h-2). \quad (17)$$

Now, it remains to clarify the estimation of N_{h-1}, which is different from the other intermediate levels on two aspects: 1) There is no LuP for the higher level. 2) The number of live entries in the root node increases up to $\lceil N/(f_l \cdot f_{nl}^{h-2})\rceil$. Following the analysis of N_i,

$$N_{h-1} = \sum_{j=2}^{\lceil N/(f_l \cdot f_{nl}^{h-2})\rceil} \frac{(N - L_{h-1})/\left(\lceil N/(f_l \cdot f_{nl}^{h-2})\rceil - 2\right)}{B_{nl} - j}. \quad (18)$$

Replacing variables in (13) correspondingly with results in (10) to (18), we obtain the cost model that predicts the structure size of the aP-tree. In this paper, we assume that each disk page corresponds to one structure node; hence, the model also gives the number of pages required by an aP-tree. It is straightforward to extend the equation to the general case where a node corresponds to multiple disk pages. The estimation for query costs is relatively simple. As discussed in the previous section, processing a VRA query involves visiting at most two paths from the root to the leaf level of a B-tree. Since the two paths start from the root node of the same logical B-tree, the number of node accesses in answering a VRA query is at most:

$$Cost_{aP-VRA} = 2h - 1 = 3 + 2\left\lceil \log_{f_{nl}} \frac{N/f_l}{SP_{nl}}\right\rceil. \quad (19)$$

Thus, the cost of answering a RA query (i.e., two VRA queries) is given by (20). The formula involves a very low constant value irrespective of the sizes and positions of the queries.

$$Cost_{aP-RA} = 2Cost_{aP-VRA} = 6 + 4\left\lceil \log_{f_{nl}} \frac{N/f_l}{SP_{nl}}\right\rceil. \quad (20)$$

4.4 Making the aP-Tree Dynamic

As mentioned in Section 2.3, the multiversion framework does not support insertions to the "history." As a result, the aP-tree allows only insertions in ascending order of data points' x-coordinates. If this is not true (i.e., in practice, new points do not necessarily arrive in this order), the aP-tree becomes static. When there are no deletions (all points ever inserted will exist forever), we can make the aP-tree (semi)dynamic using the *external logarithmic method* [5], a general technique that enables insertions to a static index. In the fully dynamic scenario (i.e., with deletions), however, the logarithmic method is inapplicable as it requires the original structure to support deletions, which is not true for the aP-tree (and the multiversion framework in general).

To solve this problem and make the aP-tree fully dynamic, we propose the *double logarithmic method*. Given an initial data set with N points, at the initial step, we create $\log_B N$ aP-trees (denoted as $I_1, I_2, \ldots, I_{\log_B N}$ and collectively referred to as a *structure set*) in the same way as the logarithmic method, so that the number $|I_i|$ of points in the ith $(1 \le i \le \log_B N)$ tree is no more than B_i points. The insertion of each point involves 1) identifying the jth tree such that j is the smallest integer satisfying $\sum_{i=1}^{j} |I_i| \le B_j$, 2) discarding all the structures $I_1, I_2, \ldots, I_j$, and 3) building a new I_j from the points in the discarded structures using the bulkloading algorithm of Section 4.2 in

$$O(B_j/B \log_B(N/B)) = O(B_{j-1} \log_B(N/B))$$

I/Os (after this step, $I_1, I_2, \ldots, I_{j-1}$ become empty). Since we move at least $\sum_{i=1}^{j-1} |I_i| > B_j - 1$ (recall the way j was chosen) points to I_j, the amortized construction cost per point becomes $O(\log_B(N/B))$ I/Os. Further, given the fact that, a point is always moved to a tree with larger subscript (i.e., moved at most $O(\log_B(N/B))$ times), the total insertion overhead per point is $O(\log_B^2(N/B))$.

Insertions after the initial step are performed in exactly the same way except that we keep the total number N_I of insertions (to be used later). To delete a point p, instead of removing it from the structure set $\{I_1, I_2, \ldots, I_{\log_B N}\}$ (as is the case for the original logarithmic method), we actually insert it to *another* structure set, which also contains $\log_B N$ aP-trees $D_1, D_2, \ldots, D_{\log_B N}$, in the same way as described earlier for I_i. Therefore, the total deletion overhead per point is $O(\log_B^2(N/B))$. We also keep track of the total number N_D of deletions. Finally, when $N_I + N_D = N/2$, we destroy *all* the trees and perform a global rebuilding [5] with the *remaining* points, i.e., those that are present in I_i but not in

D_i (note that these points can be easily maintained by a separate B-tree on their IDs). Specifically, we create a single aP-tree $I_{\log_B N}$, which needs $O((N/B)\log_B(N/B))$ I/Os and, thus, charging each of the $(N/2)$ insertions/deletions $O((1/B)\log_B(N/B)) = O(\log_B^2(N/B))$ I/Os.

Given a query, we first obtain the partial result PI_i from each I_i, and PD_i from each $D_i (1 \leq i \leq \log_B N)$ and compute the final result as $\sum_i PI_i - \sum_i PD_i$. Since querying each tree costs $O(\log_B N/B)$, the total cost is $O(\log_B^2(N/B))$. Notice that, the result is correct even if a single point is inserted/deleted multiple times, in which case there will be multiple copies in I_i and D_i. Finally, the double logarithmic method is general and applies to all problems satisfying the following property: The problem is *decomposable* [5] and the final result can be obtained from the partial ones from the insertion/deletion set in $O(1)$ time.[8] For instance, this technique can be applied to make the MVSB-tree [40] fully dynamic. We close the section with the following theorem that summarizes the update and query cost of dynamic aP-trees.

Theorem 3. *Given N 2D points, a dynamic aP-tree consumes $O((N/B)log_B(N/B))$ space, can be incrementally maintained in amortized $O(log_B^2(N/B))$ I/Os per insertion/deletion and answers a RA query $O(log_B^2(N/B))$ I/Os.*

4.5 Three-Dimensional Aggregate Trees

The aR-tree can be used for any dimensionality d, however, suffering from the high query cost, which increases with d as explained in Section 2.1. In order to alleviate the problem, we develop a 3D structure that combines the properties of aP and aR -trees. Fig. 12 motivates the concept; a 3D point (x, y, z) can be thought of as an interval whose x-projection is $[x, X]$ (X is the maximum x-coordinate), and whose projection onto the y-z plane is point (y, z). Accordingly, a RA query q is still reduced to two VRA queries q_0, q_1 each of which retrieves the number (2 and 4 for q_0, q_1, respectively) of data intervals crossing a 2D rectangle vertical to the x-axis. The output of q equals the difference (i.e., 2) of the results of q_0, q_1. Thus, we can solve the problem by maintaining a (logical) structure (e.g., the aR-tree or the 2D aP-tree) optimized for 2D RA search at every x-coordinate in the 3D space.[9] The various logical structures are managed by the modified multiversion framework of the aP-tree, which permits aggregate retrieval. Particularly, if the x-coordinate of the new point is larger than the starting timestamp of an intermediate entry on the insertion path, the entry is duplicated and its aggregate number incremented.

As the logical structure at each timestamp, we adopt the 2D aR-tree because deploying 2D aP-trees leads to a complex multilayered structure that is less interesting in practice. The performance analysis of the resulting 3D tree follows that of the previous sections. First, its space complexity is $O((N/B)\log_B(N/B))$, and it can be bulkloaded in

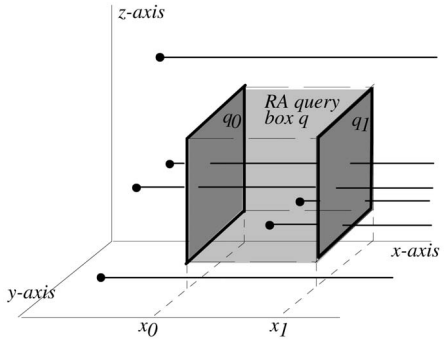

Fig. 12. Reduction of RA queries to VRA in 3D.

$O((N/B)\log_B(N/B))$ I/Os, where N is the data set cardinality and B the node capacity. Its asymptotical query cost is identical to that of the 2D aR-tree, or specifically $O((N/B)^{1/2})$ I/Os. Using the double logarithmic technique presented in the previous section, the 3D tree can be made fully dynamic, supporting insertions and deletions in amortized $O(\log_B^2(N/B))$ I/Os per operation, while degrading the search performance to $O((N/B)^{1/2}\log_B(N/B))$. Regarding practical cost, the space consumption can also be computed using (13) (applying (14) to (18)), and the number of node accesses in processing a 3D RA query is (at most) twice that of a 2D aR-tree, which is represented in (6). The same technique can be applied for higher dimensions, i.e., a d-dimensional aggregate tree can be obtained by a multiversion $(d-1)$-dimensional aR-tree, using our framework for converting an aggregate index into its multiversion counterpart. The resulting structure improves the d-dimensional aR-tree by a factor of $O((N/B)^{1/[d(d-1)]})$, but obviously the cost savings decrease with d.

5 Experiments

In this section, we compare the performance of aP and aR-trees using synthetic and real data sets in two and three-dimensional spaces (all axes are normalized to unit length). Since we deal with the *count* aggregate, we only keep aggregate information at intermediate levels. For aP-trees, the x_{end} field of a leaf entry (i.e., the ending timestamp of its lifespan) is not stored (as discussed in Section 4 it is always *). The aR-implementation is based on the R*-tree [8]. The page size is set to 4k bytes, so that the leaf and intermediate node capacities of the aP-tree are 255 (204) and 204 (127), respectively, in the 2D (3D[10]) space. The corresponding numbers for aR-trees are 255 (204) and 170 (127). The P_{svo} parameter of the aP-tree is set to 0.5 in all cases, i.e., a version split is followed by a key split if the new node is at least half full. Performance is measured as the average number of node accesses (NA) in answering a workload of 500 RA queries. Each query hyper-rectangle in the workload has the same length q_L on each dimension and its location follows the data distribution.

8. An ordinary range search (i.e., finding the actual objects), for example, does not satisfy this property, in which case a *set difference* operation must be performed.

9. A similar idea [30], proposed for the MOLAP model (i.e., highly dense data on a multidimensional array), is inapplicable to our case because it assumes 1) discrete axes and 2) that the data cardinality increases exponentially with the dimensionality. On the other hand, our solution is more general as it applies to the MOLAP model.

10. The 3D aP-tree corresponds to the structure presented in Section 4.5.

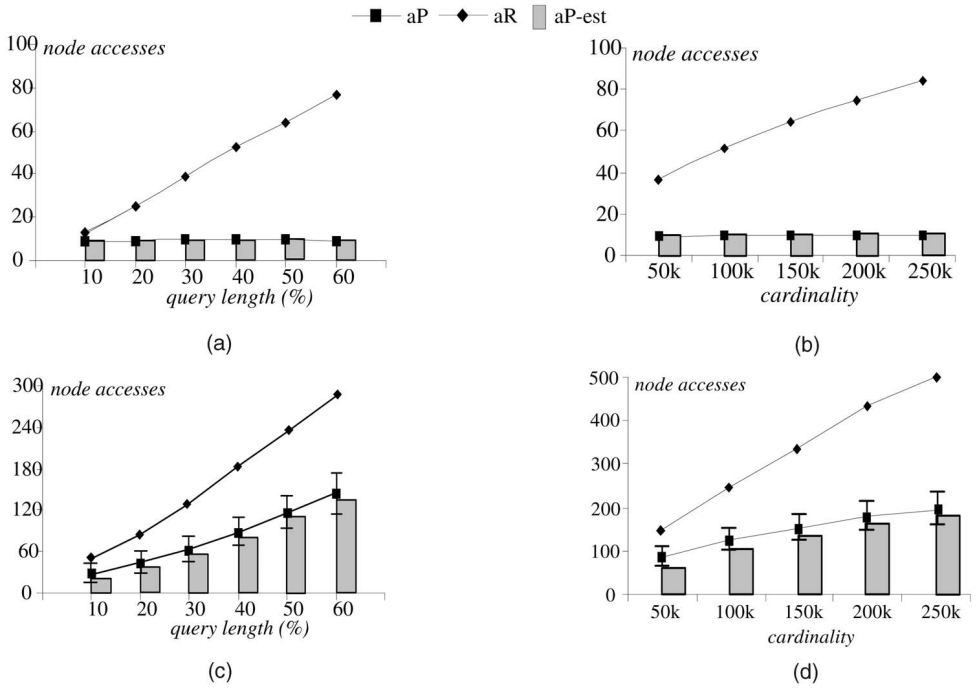

Fig. 13. Node accesses versus query length q_L (uniform data). (a) NA versus q_L ($N = 150$k, 2D). (b) NA versus N ($q_L = 50$ percent, 2D). (c) NA versus q_L ($N = 150$k, 3D). (d) NA versus N ($q_L = 50$ percent, 3D).

5.1 Uniform Data Sets

We first evaluate the aP and aR-trees using uniform data sets with various cardinalities. Fig. 13a plots the query cost as a function of the query length q_L (ranging from 10 to 60 percent of the axis) for aP and aR-trees indexing 150k 2D points, together with the estimated costs for the aP-tree (using (20)). The cost of the aR-tree increases linearly with q_L, while that of the aP-tree is constant and considerably lower (e.g., for 60 percent query length, the aP-tree is more than eight times faster). The estimates are exactly the same as the actual values (In particular, since the height of the aP-tree is 3, its actual cost equals 10 node accesses), confirming the correctness of our derivation.

Fig. 13b shows the node accesses (using queries with length 50 percent) as a function of the cardinality N for uniform data sets with 50k, 100k, 150k, 200k, and 250k points. The performance of the aR-tree deteriorates quickly with N, while that of the aP-tree remains constant since there is no change in the tree height. Figs. 13c and 13d repeat the above experiments in the 3D space (including the maximum and minimum query costs). In this case, our model yields a small error (less than 10 percent) due to the fact that the aR-tree performance cannot be fully predicted (recall that, our model in the 3D case utilizes the previous results on aR-trees). The cost of the (3D) aP-tree now increases with the query length because as mentioned in Section 4.5; in this case, two 2D aR-trees (whose costs increase with q_L) should be searched.

As discussed in Section 4.3, the improvement of aP-trees comes at the expense of extra space consumption. To evaluate the trade-off between space and query cost, Fig. 14a (14b) compares the sizes of aP and aR-trees as a function of the data set cardinality in the 2D (3D) space. As expected, the aR-tree requires less space which, however, does not justify its high query overhead (especially in 2D). Interestingly, despite the fact that the size complexity of the aP-tree is $O((N/B)\log_B(N/B))$, its growth is quite linear for the cardinalities inspected. This happens because the factor $\log_B(N/B)$ actually corresponds to the height of the tree. Therefore, the size of the aP-tree grows linearly as long as its height remains constant, which is true in Figs. 14a and 14b.

5.2 Nonuniform Data Sets

In this section, we compare the aR and aP-trees using six nonuniform data sets described as follows:

1. *CFD1* (52k points) are vertex data from various Computation Fluid Dynamic models.[11]
2. *CFD2* (200k points) are vertex data from various Computation Fluid Dynamic models.
3. *SCG* contains 62k points representing gravity data[12] in California.
4. SCP contains 46k points corresponding to places in South California.[13]
5. *gauss* contains 100k 3D points following the Gaussian distribution.
6. *zipf* has the same cardinality and dimensionality, but the distribution is Zipf.

As with the uniform case, we start with the query cost comparison. Fig. 15 illustrates the number of node accesses as a function of the query length q_L for all data sets.

The aP-tree outperforms the aR-tree in all cases and the improvement increases with q_L (up to an order of

11. Available at http://www.cs.du.edu/~leut/MultiDimData.html.
12. Available at http://www.gps.caltech.edu/~clay/gravity/gravity.html.
13. Available at http://dias.cit.gr/~ytheod/research/datasets/spatial.html.

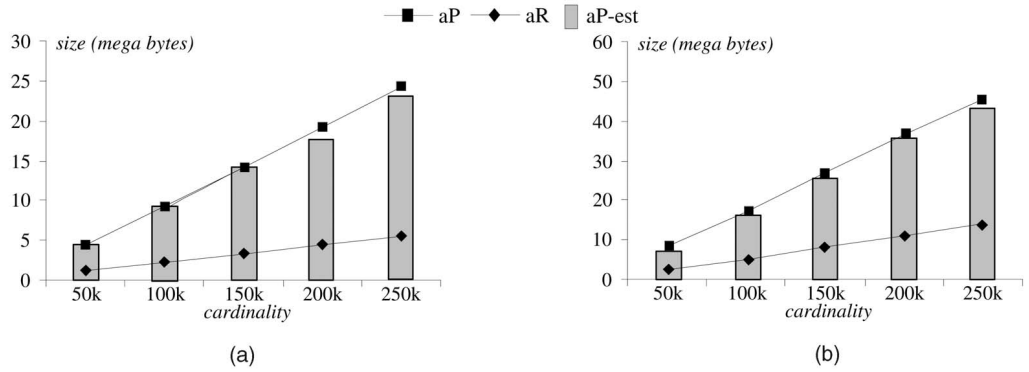

Fig. 14. Structure size versus data set cardinality N (uniform data). (a) Size versus N (2D). (b) Size versus N (3D).

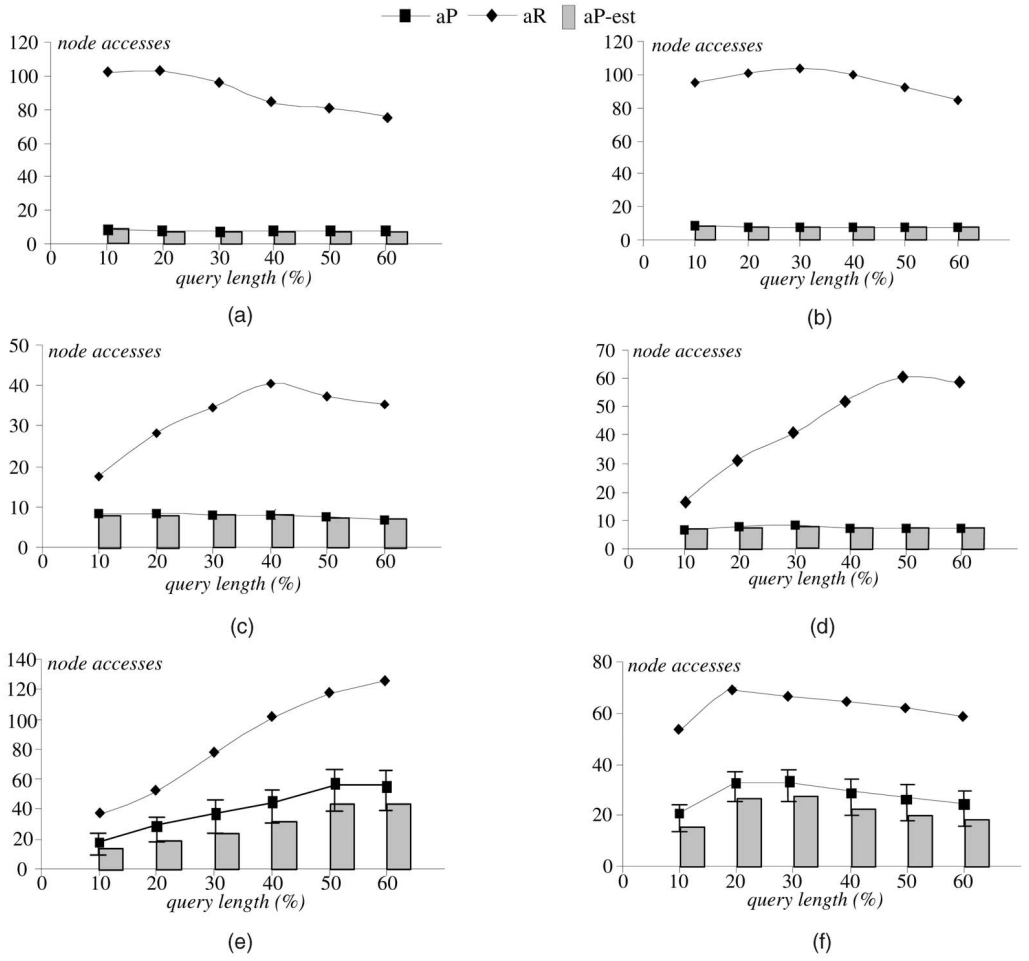

Fig. 15. Node accesses versus query length q_L (nonuniform). (a) NA versus q_L (CFD1). (b) NA versus q_L (CFD2). (c) NA versus q_L (SCG). (d) NA versus q_L (SCP). (e) NA versus q_L (3D Gaussian). (f) NA versus q_L (3D zipf).

magnitude). An interesting observation is that, for nonuniform distributions, the cost of the aR-tree usually drops when the query length grows beyond a certain threshold. This happens because the node MBBs for these data sets are more skewed and, if the query hyper-rectangle is sufficiently large, most node MBBs fall inside the query (i.e., they do not intersect the boundary) and are not visited. This phenomenon does not exist for uniform distributions, where node MBBs spread evenly across the data space. For 2D data, the overhead of the aP-tree is again independent of q_L; for 3D data (Figs. 15e and 15f), its cost demonstrates similar behavior to that of the aR-tree. Our model is precise for the 2D case, while yielding error below 20 percent for the 3D case (as with Figs 13c and 13d, we also plot the minimum/maximum query cost). Finally, Fig. 16 demonstrates the sizes of the aP and aR-trees for the above data sets, as well as the estimated values from (13) (for aP-trees). The difference between the two structures is similar to that of the uniform case.

In summary, we have experimentally demonstrated the efficiency of the proposed aP-tree, which outperforms the aR-tree by up to an order of magnitude. Particularly, the

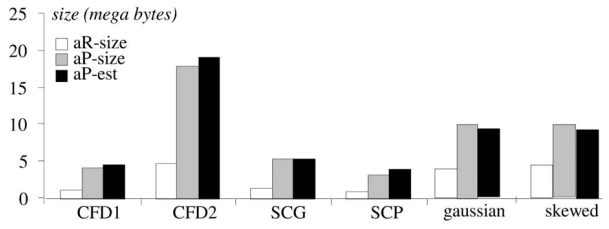

Fig. 16. Structure sizes for nonuniform data sets.

aP-tree is especially efficient for the two-dimensional RA query (where its query cost is constant), which is imperative to a large number of practical applications. Further, we confirm the accuracy of the cost models for predicting the query cost (average error 5/20 percent for uniform/nonuniform data) and space consumption (5 percent error in all cases).

6 Conclusions

This paper presents a detailed study on range aggregate queries in spatial databases. We first provide a complete analysis for the performance of the aR-tree revealing its inefficiency. Then, we propose the aP-tree and prove, both analytically and experimentally, that the aP-tree consistently outperforms the aR-tree. Further, compared with the existing theoretical indexes, the aP-tree has significantly higher applicability because it is simple and accompanied by accurate cost models for space consumption and query performance. An open problem is to extend the aP-tree for supporting RA queries on rectangular data. Theoretically, as shown in [16], any method for RA retrieval on point data also solves rectangles with the same asymptotical performance, by maintaining, however, a large number of structures. A more practical solution with lower space requirements is highly desirable (a pioneering work, yet to be improved, can be found in [41]). Further, we are not aware of any existing work dealing with exact aggregate computation of joins between multiple data sets (e.g., retrieve the total number of (restaurant, hotel) pairs that are within one mile from each other). Handling such queries effectively may require novel access methods and query algorithms. Finally, deploying similar ideas to other aggregates (e.g., *max* as in [4]) is also interesting.

Acknowledgments

This work was supported by grants CITYU 1163/04E and HKUST 6197/02E from the Hong Kong Research Grants Council.

References

[1] L. Arge, "External Memory Data Structures," *Handbook of Massive Data Sets*, J. Abello, P.M. Pardalos, M.G.C. Resende, eds., pp. 313-357, Kluwer Academic Publishers, 2002.

[2] L. Arge, "The Buffer Tree: A New Technique for Optimal I/O-Algorithms," *Proc. Workshop Algorithms and Data Structures*, 1995.

[3] S. Acharya, V. Poosala, and S. Ramaswamy, "Selectivity Estimation in Spatial Databases," *Proc. SIGMOD Conf.*, 1999.

[4] P. Agarwal, L. Arge, J. Yang, and K. Yi, "I/O-Efficient Structures for Orthogonal Range Max and Stabbing Max," *Proc. European Space Agency Conf.*, 2003.

[5] L. Arge and J. Vahrenhold, "I/O-Efficient Dynamic Planar Point Location," *Proc. ACM Symp. Computational Geometry*, 2000.

[6] A. Belussi and C. Faloutsos, "Estimating the Selectivity of Spatial Queries Using the Correlation's Fractal Dimension," *Proc. Very Large Databases Conf.*, 1995.

[7] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An Asymptotically Optimal Multiversion B-Tree," *Very Large Databases J.*, vol. 5, no. 4, pp. 264-275, 1996.

[8] B. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method," *Proc. SIGMOD Conf.*, 1990.

[9] J. Bercken, B. Seeger, and P. Widmayer, "A Generic Approach to Bulk Loading Multidimensional Index Structures," *Proc. Very Large Databases Conf.*, 1997.

[10] C. Chung, S. Chun, J. Lee, and S. Lee, "Dynamic Update Cube for Range-Sum Queries," *Proc. Very Large Databases Conf.*, 2001.

[11] C. Chan and Y. Ioannidis, "Hierarchical Cubes for Range-Sum Queries," *Proc. Very Large Databases Conf.* , 1999.

[12] M. Denny, M. Franklin, P. Castro, and A. Purakayastha, "Mobiscope: A Scalable Spatial Discovery Service for Mobile Network Resources," *Mobile Data Management*, 2003.

[13] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, "Making Data Structures Persistent," *J. Computer and System Sciences*, vol. 38, no. 1, pp. 86-124, 1989.

[14] C. Faloutsos and I. Kamel, "Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension," *ACM Symp. Principles of Database Systems*, pp. 4-13, 1994.

[15] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. SIGMOD Conf.*, 1984.

[16] S. Govindarajan, P. Agarwal, and L. Arge, "CRB-Tree: An Efficient Indexing Scheme for Range Aggregate Queries," *Proc. Int'l Conf. Database Theory*, 2003.

[17] S. Geffner, A. Agrawal, A. Abbadi, and T. Smith, "Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes," *Proc. Int'l Conf. Data Eng.*, 2000.

[18] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tabs and Subtotals," *Proc. Int'l Conf. Data Eng.*, 1996.

[19] D. Gunopulos, G. Kollios, V. Tsotras, and C. Domeniconi, "Approximate Multi-Dimensional Aggregate Range Queries over Real Attributes," *Proc. SIGMOD Conf.*, 2000.

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

[21] M. Jurgens and H. Lenz, "The Ra*-Tree: An Improved R-Tree with Materialized Data for Supporting Range Queries on OLAP-Data," *Proc. DEXA Workshop*, 1998.

[22] M. Jurgens and H. Lenz, "PISA: Performance Models for Index Structures with and without Aggregated Data," *Proc. Int'l Conf. Statistical and Scientific Database Management*, 1999.

[23] I. Kamel and C. Faloutsos, "On Packing R-Trees," *Proc. Int'l Conf. Information and Knowledge Management*, 1993.

[24] G. Kollios, D. Gunopulos, V. Tsotras, A. Delis, and M. Hadjieleftheriou, "Indexing Animated Objects Using Spatiotemporal Access Methods," *IEEE Trans. Knowledge and Data Eng.*, vol. 13, no. 5, pp. 758-777, Sept./Oct. 2001.

[25] A. Kumar, V. Tsotras, and C. Faloutsos, "Design Access Methods for Bi-temporal Databases," *IEEE Trans. Knowledge and Data Eng.*, vol. 10, no. 1, pp. 1-20, Jan./Feb. 1998.

[26] I. Lazaridis and S. Mehrotra, "Progressive Approximate Aggregate Queries with a MultiResolution Tree Structure," *Proc. SIGMOD Conf.*, 2001.

[27] M. Muralikrishna and D. DeWitt, "Equi-Depth Histograms for Estimating Selectivity Factors for MultiDimensional Queries," *Proc. SIGMOD Conf.*, 1988.

[28] Y. Poosala and Y. Ioannidis, "Selectivity Estimation without the Attribute Value Independence Assumption," *Proc. Very Large Databases Conf.*, 1997.

[29] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient OLAP Operations in Spatial Data Warehouses," *Proc. Symp. Spatial and Temporal Databases*, 2001.

[30] M. Riedewald, D. Agrawal, and A. Abbadi, "Efficient Integration and Aggregation of Historical Information," *Proc. SIGMOD Conf.*, 2002.

[31] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-Tree: A Dynamic Index for MultiDimensional Objects," *Proc. Very Large Databases Conf.,* 1987.
[32] B. Salzberg and V. Tsotras, "A Comparison of Access Methods for Temporal Data," *ACM Computing Surveys,* vol. 31, no. 2, pp. 158-221, 1999.
[33] Y. Tao and D. Papadias, "The MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries," *Proc. Very Large Databases Conf.,* 2001.
[34] Y. Tao, D. Papadias, and J. Zhang, "Efficient Cost Models for Overlapping and MultiVersion Structures," *ACM Trans. Database Systems,* vol. 27, no. 3, pp. 299-342, 2002.
[35] Y. Theodoridis and T. Sellis, "A Model for the Prediction of R-tree Performance," *Proc. Symp. Principles of Database Systems,* 1996.
[36] P. Varman and R. Verma, "An Efficient Multiversion Access Structure," *IEEE Trans. Knowledge and Data Eng.,* vol. 9, no. 3, pp. 391-409, May/June 1997.
[37] J. Vitter and M. Wang, "Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets," *Proc. SIGMOD Conf.,* 1999.
[38] A. Yao, "Random 2-3 Trees," *Acta Informatica,* vol. 2, no. 9, pp. 159-179, 1978.
[39] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates," *Proc. Int'l Conf. Data Eng.,* 2001.
[40] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger, "Efficient Computation of Temporal Aggregates with Range Predicates," *Proc. Symp. Principles of Database Systems,* 2001.
[41] D. Zhang, V. Tsotras, and D. Gunopulos, "Efficient Aggregation over Objects with Extent," *Proc. Symp. Principles of Database Systems,* 2002.

Yufei Tao received the diploma from the South China University of Technology in 1999, and the PhD degree from the Hong Kong University of Science and Technology in 2002, both in computer science. After that, he spent a year as a visiting scientist at the Carnegie Mellon University. Currently, he is an assistant professor in the Department of Computer Science at the City University of Hong Kong. He is the winner of the Hong Kong Young Scientist Award 2002 from the Hong Kong Institution of Science. His research includes query algorithms and optimization in temporal, spatial, and spatio-temporal databases.

Dimitris Papadias is an associate professor in the Computer Science Department at the Hong Kong University of Science and Technology (HKUST). Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and the University of Patras (Greece). He has published extensively and been involved in the program committees of all major database conferences, including SIGMOD, VLDB, and ICDE.