Aggregate Processing of Planar Points

Yufei Tao, Dimitris Papadias, Jun Zhang

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{taoyf, dimitris, zhangjun@cs.ust.hk}

Abstract. Aggregate window queries return summarized information about objects that fall inside a query rectangle (e.g., the number of objects instead of their concrete ids). Traditional approaches for processing such queries usually retrieve considerable extra information, thus compromising the processing cost. The paper addresses this problem for planar points from both theoretical and practical points of view. We show that, an aggregate window query can be answered in logarithmic worst-case time by an indexing structure called the aP-tree. Next we study the practical behavior of the aP-tree and propose efficient cost models that predict the structure size and actual query cost. Extensive experiments show that the aP-tree, while involving more space consumption, accelerates query processing by up to an order of magnitude compared to a specialized method based on R-trees. Furthermore, our cost models are accurate and can be employed for the selection of the most appropriate method, balancing the space and query time tradeoff.

1. Introduction

Window queries retrieve the objects that fall inside (or intersect) a multi-dimensional window. Such queries are important for numerous domains and have been studied extensively in the database literature. Recently a related type, called the *window aggregate* query (WA for short), is gaining increasing attention in the context of OLAP applications. A WA query returns summarized information about objects that fall inside the query window, for example the number of cars in a road segment, the average number of mobile phone users per city block etc. An obvious approach to answer such queries is to first retrieve the actual objects by performing traditional window queries, and then compute the aggregate function. This, however, entails a lot of unnecessary effort, compromising performance. A solution for the problem is to store aggregate information in the nodes of specialized index structures. Such aggregate trees have already been employed in the context of temporal databases for computing aggregates over temporal data [KS95, KKK99, YW01, ZMT+01].

In order to improve the performance of WA queries in OLAP applications involving multi-dimensional ranges, Jurgens and Lenz [JL98] proposed the storage of summarized data in the nodes of the R-tree [BKS+90] used to index the fact table. The same concept was applied in [PKZ+01] for spatial data warehouses. Each entry of the resulting *aggregate R-tree* (*aR-tree*), in addition to the minimum bounding rectangle

(MBR), stores summarized data about objects under the corresponding subtree. As a result, nodes totally contained by the query window do not have to be accessed during the processing of WA queries. aR-trees and other aggregate multi-dimensional indexes were employed by [LM01] in order to compute fast approximate answers of OLAP queries. By traversing the index, a rough approximation is obtained from the values at the higher levels, which is progressively refined as the search continues towards the leaves. Papadias et al [PTK+02] proposed combinations of aggregate R- and B-trees for indexing spatio-temporal data warehouses.

Although aR-trees (and other aggregate trees based on the straightforward adaptation of multi-dimensional structures) improve performance of WA queries considerably compared to regular R-trees, their processing cost can still be very high, especially for queries with large ranges common in OLAP applications. In this paper, we focus on aggregate processing of planar points and show that any WA query can be answered with $O(\log_b n)$ page accesses by a specialized indexing structure, the *aggregate Point-tree (aP-tree)*, which consumes $O(n/b \log_b n)$ space, where n is the number of data points and b the disk page size. The intuition behind aP-trees, is that two-dimensional points can be viewed as intervals in the key-time plane and indexed by temporal access methods.

In addition to asymptotic performance, we analyze the practical behavior of aP-trees, and propose cost models for their sizes and query costs. Extensive experimentation shows that the aP-tree is more than just theoretical contribution, since it answers WA queries significantly faster than the aR-tree while consuming some more space. Besides their applicability in traditional OLAP applications, aP-trees are important for spatial [PT01] and spatio-temporal [PTK+02] data warehouses.

The rest of the paper is organized as follows. Section 2 surveys the aR-tree and proposes cost models for query performance. It also introduces the multi-version B-tree, which provides the main motivation for the aP-tree. Section 3 discusses the aP-tree in detail and proves its asymptotical performance. Section 4 contains cost models that accurately predict the structure size and query cost of aP-trees. Section 5 presents an extensive experimental evaluation with synthetic and real datasets, and Section 6 concludes the paper with directions for future work.

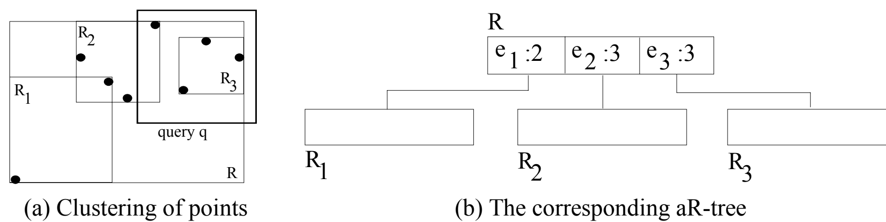
2. Related Work

Existing work on aggregate trees has been based mostly on the R-tree due to its popularity as a multi-dimensional access method. In this section, we describe the aR-tree and analyze its expected performance on WA queries. Next we overview the multi-version B-tree and its related algorithms.

2.1 The aggregate R-tree (aR-tree) and analysis

The aggregate R-tree improves the original R-tree towards aggregate processing by storing, in each intermediate entry, summarized data about objects residing in the subtree. In case of the COUNT function, for example, each entry stores the number of

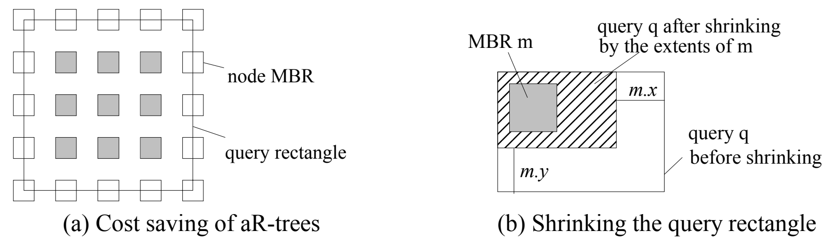
objects in its subtree (the extension to any non-holistic functions is straightforward). Figure 1a shows a simple example where 8 points are clustered into 3 leaf nodes R_1, R_2, R_3, which are further grouped into a root node R. The solid rectangles refer to the MBR of the nodes. The corresponding R-tree with intermediate aggregate numbers is shown in Figure 1b. Entry e_1:2, for instance, means that 2 points are in the subtree of e_1 (i.e., node R_1). Notice that each point is counted only once, e.g., the point which lies inside the MBRs of both R_1 and R_2 is added to the aggregate result of the node where it belongs (e_1). The WA query represented by the bold rectangle in Figure 1a is processed in the following manner. First the root R is retrieved and each entry inside is compared with the query rectangle q. One of the 3 following conditions holds: (i) the (MBR of the) entry does not intersect q (e.g., entry e_1) and its sub-tree is not explored further; (ii) the entry partially intersects q (e.g., entry e_2) and we retrieve its child node to continue the search; (iii) the entry is contained in q (e.g., entry e_3), in which case, it suffices to add the aggregate number of the entry (e.g., 3 stored with e_3) without accessing its subtree. As a result, only two node visits (R and R_2) are necessary. Notice that conventional R-trees would require 3 node visits.



(a) Clustering of points (b) The corresponding aR-tree

Fig. 1. An aR-tree example

In summary, the improvement of the aR-tree over the conventional R-tree is that we do not need to visit the nodes (whose MBRs are) inside the query window, but only those nodes that intersect the edges of the window. To answer the query in Figure 2a (where the nodes correspond to the clustering on uniform points as in [TS96]), for example, the aR-tree only needs to retrieve the white nodes, while the R-tree must also access the gray ones. The cost savings obviously increase with the size of the query window, an important fact because OLAP queries often involve large ranges. Notice, however, that despite the improvement of the aR-tree, query performance is still sensitive to the window size since, the larger the window, the higher the number of node MBRs that are expected to intersect its sides.



(a) Cost saving of aR-trees (b) Shrinking the query rectangle

Fig. 2. Cost analysis of aR-trees

This sensitivity of aR-trees to window area, is confirmed by the following analysis. For a query q and a node MBR m, let $PR_{intr}(q,m)$, $PR_{part}(q,m)$, and $PR_{cont}(q,m)$ refer to the probabilities that q intersects, partially intersects, and contains m respectively:

$$PR_{intr}(q,m) = PR_{part}(q,m) + PR_{cont}(q,m)$$

The estimation of $PR_{intr}(q, m)$ was previously studied in [KF93]. Assuming a unit universe and uniform distribution of points, $PR_{intr}(q, m)$ can be represented as

$$PR_{intr}(q,m) = (q.x + m.x) \cdot (q.y + m.y)$$

where $q.x$ and $q.y$ refer to the lengths of the horizontal and vertical extents of q respectively (and similarly for $m.x$ and $m.y$). On the other hand, observe that query q contains m if and only if, after shrinking q from the lower-right corner point by the corresponding extents of m along each dimension, the resulting rectangle still contains the upper-left corner point of m, as illustrated in Figure 2b. Hence, the probability that the query window contains m is:

$$PR_{cont}(q,m) = (q.x - m.x) \cdot (q.y - m.y)$$

So the probability of partial intersection is:

$$PR_{part}(q,m) = PR_{intr}(q,m) - PR_{cont}(q,m) = 2(m.x \cdot q.y + m.y \cdot q.x)$$

$PR_{part}(q, m)$ corresponds to the probability that a node with MBR m is visited in answering the WA query q using the aR-tree. Let h be the height of the aR-tree, N_i the number of nodes at level i, and s_i the average MBR of nodes at level i; then, the expected number of node accesses in answering q is:

$$NA(q) = \sum_{i=0}^{h-1} [N_i \cdot PR_{part}(q, s_i)]$$

The estimation for h, N_i, and s_i was studied in [TS96], where the following results were obtained (f_R is the average fanout):

$$h = \left\lfloor \log_{f_R} n \right\rfloor,\ N_i = n / f_R^{\,i+1},\ \text{and}\ s_{i1} = s_{i2} = \sqrt{D_{i+1} \frac{f_R^{\,i+1}}{n}} \quad (0 \le i \le h-1)$$

$$\text{where}\ D_{i+1} = \left[1 + \frac{\sqrt{D_i} - 1}{\sqrt{f_R}}\right]^2 \text{and}\ D_0 = 0$$

Therefore the cost of processing a WA query q with aR-trees is:

$$NA(q) = \sum_{i=0}^{\left\lfloor \log_{f_R} n \right\rfloor - 1} [2\sqrt{\frac{D_{i+1} \cdot n}{f_R^{\,i+1}}}(q.x + q.y)] \quad (2.1)$$

It is clear that the cost increases with the lengths of the query's extents and can be prohibitive for large query windows. This is a serious problem because aR-trees were motivated by the need to efficiently process queries with large windows in the first place. aP-trees overcome this problem (i.e., the cost is independent of the query extent) by transforming points to intervals in the key-time plane and adapting specialized interval indexing methods such as the multi-version B-tree introduced next.

2.2 The Multi-version B-tree (MVB-tree)

The multi-version B-tree [BGO+96] is an extension of the B-tree for indexing the evolution of one-dimensional data in *transaction time temporal databases* [ST97],

which are best described as intervals in the key-time plane. In the example of Figure 3, intervals a_1, a_2, a_3, and b correspond to the bank balances of two accounts a and b. The key axis represents the amount of the balance. Both accounts are created at time t_0 and cancelled at t_3. There are two changes to account a: one withdrawal at t_1 and one deposit at t_2, while account b remains constant during the period $[t_0, t_3)$. In the sequel, we represent an interval (e.g., a_1) in the form *lifespan*: *key* (e.g., $[t_0, t_1)$:a_1).

Intervals are inserted into a MVB-tree in a plane-sweep manner. To be specific, at the beginning a vertical sweeping line is placed at the starting point of the left-most interval before it starts moving right. An interval is inserted when its starting point is reached by the sweeping line. For example, at t_0, a_1 and b are inserted as $[t_0$, *):a_1 and $[t_0$, *):b respectively, where * means that the ending point of the interval is not determined yet, but progresses with the sweeping line (such intervals are said to be *alive*). When the ending point of an interval is encountered, its lifespan is finalized, and the interval *dies* (it is *logically deleted*). For instance, when the sweeping line reaches t_1, interval a_1 is modified to $[t_0, t_1)$: a_1.

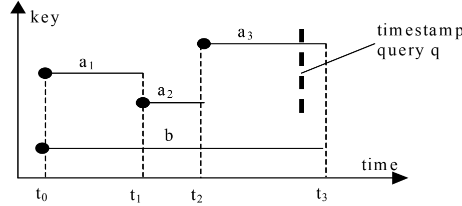


Fig. 3. Representation of temporal data

The MVB-tree is optimized for the so-called *timestamp query*, which, as shown in Figure 3, is a vertical line segment q that retrieves the horizontal line segments intersecting q. Figure 4 illustrates a simple MVB-tree when the sweeping line is at time 3. Each entry has the form <*key*, t_{start}, t_{end}, *pointer*>. For leaf entries, the *pointer* points to the actual record with the corresponding *key* value, while, for intermediate entries, it points to a next level node. The temporal attributes t_{start} and t_{end} denote the time that the record was inserted and (logically) deleted in the tree respectively.

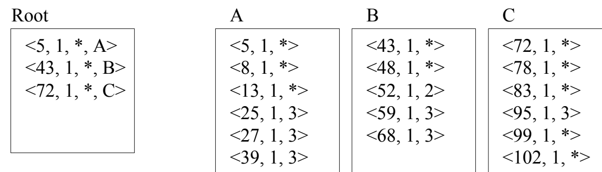


Fig. 4. Example of MVB-tree

There can be multiple logical B-trees in a MVB-tree, and each root of a logical tree has a *jurisdiction interval,* which is the minimum bounding interval of the lifespans of all the entries in the logical tree. Processing of a timestamp query starts by retrieving the corresponding root whose jurisdiction interval contains the queried timestamp, after which the search is guided by *key*, t_{start}, and t_{end}. For each timestamp t and each 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 (for the following examples $P_{version}$=1/3 and

b=6). This *weak version condition* ensures that entries alive at the same timestamps are mostly grouped together in order to facilitate timestamp query processing. Violations of this condition generate *weak version underflows*, which occur as a result of deletions.

Insertions and deletions are carried out in a way similar to B-trees except that overflows and underflows are handled differently. *Block overflow* occurs when an entry is inserted into a full node, 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 (in practice this step can be avoided since the deletion time is implied by the entry in the parent node). In Figure 5, the insertion of <28,4,*> at timestamp 4 (in the tree of Figure 4) causes node *A* to overflow. A new node *D* is 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.

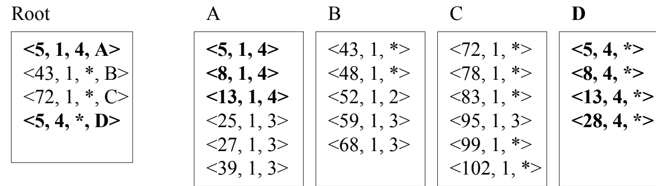


Fig. 5. Example of block overflow and version split

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 $b\cdot P_{svo}$ before the node block-overflows.

Strong version underflow is similar to weak version underflow, the only difference being that the former happens after a version split, while the latter occurs 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 Figure 4. 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, let *C′*. The insertion of <43,4,*> into *C′* causes strong version overflow, leading to a key split and, finally, nodes *D* and *E* are created. Figure 6 illustrates this process.

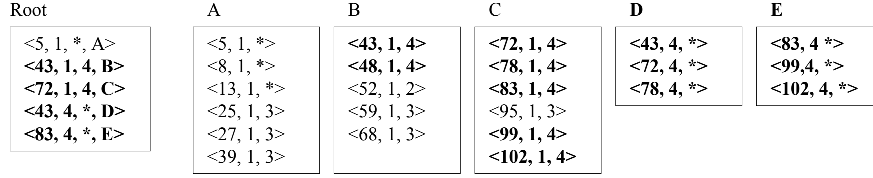
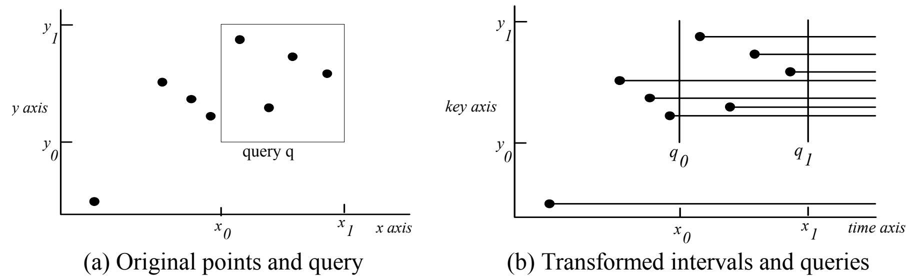


Fig. 6. Example of weak version underflow

As shown in [BGO+96], MVB-trees require $O(n/b)$ space, where n is the number of updates ever made to the database and b is the block capacity. Answering a timestamp query requires $O(\log_b m + r/b)$ node accesses, where m is the number of live intervals at the queried timestamp, and r is the number of output intervals. Both the space requirements and query performance are asymptotically optimal. A variation of MVB-trees, which reduces the tree size by a constant factor can be found in [VV97]. Several algorithms for processing interval queries and temporal joins with MVB-trees, are proposed in [BS96] and [ZTS02], respectively. The multi-version framework has also been applied to R-trees to obtain various bi-temporal and spatio-temporal access methods [KTF98, TP01, KGT]. General cost models for multi-version structures can be found in [TPZ02].

3. The aggregate Point-Tree (aP-tree)

A WA query can be formally defined as follows: given a set of points in the 2D universe $[0, M_x]$:$[0, M_y]$, retrieve the number $WA(q)$ of points contained in a rectangle $[x_0, x_1]$:$[y_0, y_1]$. In order to optimally solve the problem we start with the observation that a two-dimensional point can be transformed to an open-ended interval (in the key-time plane as described in section 2.2) as follows: the y-coordinate of the point can be thought of as a key value, while the x-coordinate represents the starting time of the interval. The ending time of all intervals is the current time (lying on the right boundary of the time axis). Figure 7a shows the points used in the example of Figure 1a, and Figure 7b illustrates the resulting intervals.



(a) Original points and query (b) Transformed intervals and queries

Fig. 7. Transformation of the problem

The original query is also transformed since the goal of query processing now is to retrieve the number of intervals that intersect the vertical line segment q_1 but not q_0. Query q_1 is represented as x_1:$[y_0, y_1]$, while q_0 as $\rightarrow x_0$:$[y_0, y_1]$. The symbol "$\rightarrow$" indicates that the x-coordinate of q_0 infinitely approaches (but does not equal) x_0 from the

left. In the sequel, we refer to q_1 and q_0 as *vertical range aggregate* (VRA) queries. In words, a VRA query x_1:[y_0, y_1] will retrieve the number $VRA(q_1)$ of intervals that start before or at timestamp x_1, and their keys are in the range [y_0, y_1]. Similarly, → x_0:[y_0, y_1] will retrieve the number $VRA(q_0)$ of intervals that start before (but not at) timestamp x_0 (this is needed in order to retrieve points at the left boundary of the query window). The result of $WA(q)$ is the (arithmetic) difference $VRA(q_1)$- $VRA(q_0)$.

Thus, WA query processing can be reduced to the vertical line segment intersection problem optimally solved by the MVB-tree, except that here we are interested in the aggregate number, instead of the concrete ids, of the qualifying objects. This fact differentiates query processing since we can avoid the retrieval of the actual objects intersecting q_1 and q_0 and the expensive computation of their set difference. In the sequel, we present the aP-tree, which modifies MVB-trees to support VRA queries.

3.1 Insertion and overflow handling

The aP-tree is similar to the MVB-tree, consisting of several logical B-trees each responsible for a jurisdiction interval, which is the minimum bounding interval of all the lifespans of its entries. An entry of the aP-tree has the form <y, [x_{start}, x_{end}), *agg*, *pointer*>, where *pointer* is the same as in MVB-trees, while y and [x_{start}, x_{end}) correspond to *key* and [t_{start}, t_{end}) as defined in section 2.2 respectively. The additional field *agg* denotes the aggregate number over the entries alive during [x_{start}, x_{end}) in the child node. Without ambiguity, in the sequel we refer to the y-field and [x_{start}, x_{end}) of each entry as the *key* and *lifespan* of the entry respectively. The minimum bounding interval of all the lifespans in a node is called the *lifespan of the node*. Figure 8a illustrates a simple example. The *agg* fields in leaf entries are omitted since they are all equal to 1 for the COUNT function. The leaf entry <5,[1,*)> in node A, for example, refers to the horizontal interval [1, M_x]:5, transformed from point (1, 5). The intermediate entry <5,[1,*),6,A> implies that there are 6 entries in node A such that (i) the key of each interval is equal to or greater than 5; (ii) the lifespan of each entry is contained in [1, M_x]. Figure 8b shows the equivalent tree where all leaf entries are represented in (p_x, p_y) format.

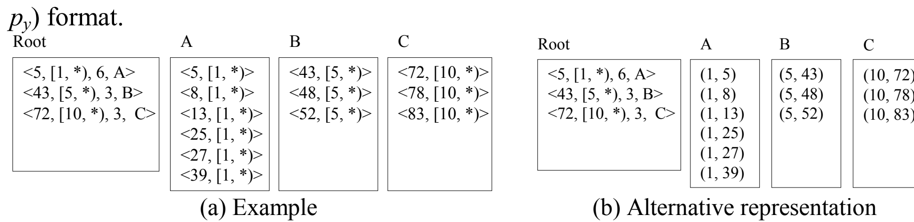


Fig. 8. An aP-tree

A preprocessing step is required to sort the original points in ascending order of their x-coordinates because, as with MVB-trees, the transformed intervals must be inserted in this order. Furthermore, since all the intervals have their right end-points on the right boundary of the universe, no logical deletion is necessary. As a result, the number of live entries in any node will never decrease until the node dies, so weak and strong version underflows never happen. Therefore, the aP-tree has only a single pa-

rameter P_{svo} (no parameters $P_{version}$ and P_{svu}), which denotes the strong version overflow threshold.

Insertion is performed as in MVB-trees except that it may be necessary to duplicate intermediate entries on the path. As an example, assume that an interval <55,[15,*)> (or equivalently point (15,55)) is inserted into the tree in Figure 8. Following the intermediate entry <45,[5,*),3,B>, the leaf node *B* that accommodates the new entry is first identified. As shown in Figure 9, the following changes are applied to the root node: (i) entry <45,[5,*),3,B> dies, having its x_{end} modified to 15; (ii) a new live entry, pointing to the same child node, is created with its x_{start} set to 15. The aggregate number of the new entry is incremented to accommodate the insertion. Such entry duplication is required by the aP-tree (but not 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 necessary when the x_{start} of the intermediate entry is smaller than the x_{start} of the interval being inserted.

Root	A	B	C
<5, [1, *), 6, A>	<5, [1, *)>	<43, [5, *)>	<72, [10, *)>
<43, [5, 15), 3, B>	<8, [1, *)>	<48, [5, *)>	<78, [10, *)>
<43, [15, *), 4, B>	<13, [1, *)>	<52, [5, *)>	<83, [10, *)>
<72, [10, *), 3, C>	<25, [1, *)>	**<55, [15, *)>**	
	<27, [1, *)>		
	<39, [1, *)>		

Fig. 9. Duplicating intermediate entry

Since no logical deletion is performed, all the entries in a leaf node must be alive when the node overflows. Therefore, a strong version overflow will always occur after version copies at the leaf level. Figure 10 illustrates a leaf overflow caused by the insertion of <40,[15,*)> (in the tree of Figure 9). Node *A* overflows and is version copied to node *D*, which generates a strong version overflow, and is finally split into itself and node *E*. Corresponding parent entries are added into the root node to reflect the changes. Notice that, due to the absence of logical deletions, the x_{end} fields of leaf entries are never modified. As an optimization method, they are not stored on the disk and this knowledge is inferred.

	Root	A	**D**	**E**	B	C
a	**<5, [1, 15), 6, A>**	<5, [1, *)>	**<5, [15, *)>**	**<25, [15, *)>**	<43, [5, *)>	<72, [10, *)>
b	**<5, [15, *), 3, D>**	<8, [1, *)>	**<8, [15, *)>**	**<27, [15, *)>**	<48, [5, *)>	<78, [10, *)>
c	**<25, [15, *), 4, E>**	<13, [1, *)>	**<13, [15, *)>**	**<39, [15, *)>**	<52, [5, *)>	<83, [10, *)>
d	<43, [5, 15), 3, B>	<25, [1, *)>		**<40, [15, *)>**	<55, [15, *)>	
e	<43, [15, *), 4, B>	<27, [1, *)>				
f	<72, [10, *), 3, C>	<39, [1, *)>				

Fig. 10. Overflow of leaf node

3.2 Vertical Range Aggregate (VRA) query processing

As discussed before, a WA query in aP-trees is reduced to two VRA queries q_1 and q_0. The processing of a VRA query x:[y_0, y_1] starts by locating the corresponding root for x, after which search is guided by the keys and lifespans of intermediate entries. For

example, in Figure 10, to answer the VRA query 15:[25, 75] we only consider those entries whose lifespans include 15; thus, entries *a* and *d* are eliminated immediately. Next, we purge the ones whose *y*-ranges do not intersect with [25, 75] (e.g., entry *b*). Among the remaining entries, some are totally inside the range [25, 75] (i.e., *c* and *e*), and for these entries, it suffices to simply add their aggregate numbers (i.e., 4 and 4). Only if the *y*-range of an entry *partially intersects* [25, 75], we have to access its child node. Continuing the example, only one leaf node *C* is visited (from entry *f*), in which one object <72,10,*> is counted. Therefore, the final answer to the query is 4 + 4 + 1 = 9.

The processing of query $\rightarrow x$:[y_0, y_1] is the same, except that intervals starting at *x* should be excluded. As an example, consider the query →15:[25, 75]. After retrieving the corresponding root node, we eliminate those entries whose intervals start at or after 15. So entries *b*, *c*, and *e* are discarded from further consideration. Then the processing proceeds as in the previous case by examining the *y*-ranges of the entries. Finally the result 8 is returned by visiting leaf nodes *A* and *C*.

An important observation is that, in the worst case, processing a VRA query only needs to visit two paths from the root to the leaf level of a B-tree. This is because, at each non-leaf level, the *y*-ranges of the entries alive at any *x*-coordinate (e.g., *x*=15 in the query example above) are continuous and disjoint. As a result, there can be at most two *y*-ranges partially intersecting the *y*-range of the query (at the start, or the end point). The *y*-ranges of the other entries are either contained by (in which cases their *agg* fields are simply added) or disjoint with the query *y*-range. Therefore, *the query cost (number of node accesses) of a WA query is at most four times the height of the B-tree* (two times for each q_1 and q_0). Since we only care about the aggregate number, it suffices to return the arithmetic difference of the two VRA queries $VRA(q_1)$-$VRA(q_0)$, without computing the set difference of their results (which would require cost up to $n\log n$, where n is the number of intervals).

3.3 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 ensured that it contains fewer than $P_{SVO}\cdot b$ entries; otherwise, a strong version overflow is generated 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:

$$N_0 \cdot (b-P_{SVO}\cdot b) \le n \Rightarrow N_0 \le \frac{n}{(1-P_{SVO})b} = O(\frac{n}{b})$$

where n is the total number of insertions (i.e., points). For a node at higher levels, the number of new entries incurred from an insertion is at most 2. This corresponds to the 2 parent entries for new nodes created at the next level (through strong version overflows). Hence, a non-leaf 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:

$$N_i \cdot (b - P_{SVO} \cdot b)/2 \le n \Rightarrow N_i \le \frac{2n}{(1-P_{SVO})b} = O(\frac{n}{b})$$

Since every node contains at least $P_{SVO} \cdot b/2 = O(b)$ entries at each point during its life-span, the height of any B-tree is $O(\log_b n)$. Therefore, the space complexity of the aP-tree is $O(n/b \log_b n)$ nodes in the worst case[1]. Furthermore, since, as with conventional B-trees, each insertion incurs at most $2\log_b n$ node accesses, the aP-tree can be constructed with $O(n\log_b n)$ node accesses, which also dominates the cost of the preprocessing sorting step.

Answering a VRA query involves visiting at most 2 paths from the root of a B-tree to the leaf level, i.e., $2\log_b n$ node accesses in the worst case. Since there are at most $O(n)$ logical B-trees (as shown later the number is very small in practice because each B-tree includes many data points), the corresponding B-tree can be located in $\log_b n$ node accesses as well, for example, by looking up a separate B-tree built on the root table. Therefore, a VRA query takes at most $O(\log_b n)$ node accesses. Given that, a WA query is transformed into 2 VRA queries, a WA query can be answered in $O(\log_b n)$ node accesses in the worst case. The following theorem summarizes the discussion above.

Theorem 3.1: A WA query for spatial points can be answered in $O(\log_b n)$ node accesses by building an aP-tree that consumes $O(n/b \log_b n)$ nodes and can be constructed with $O(n\log_b n)$ node accesses. ■

4. Cost Models for aP-trees

In the last section, we have shown theoretically that any WA query can be answered in logarithmic worst case time by introducing the aP-tree. In this section, we analyze the practical performance of the structure. This is motivated by the following facts: (i) asymptotical performance gives only limited indication towards the actual performance in practice; and (ii) the crucial factors that a database administrator needs to consider usually include the tradeoff between the size of the structure and the query response time provided. Therefore, it is important to derive accurate cost models that estimate the number of disk pages occupied by a structure, and the number of page accesses in answering a WA query.

We start by estimating the size of the aP-tree, considering, for simplicity, the case that 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 life-span. Similarly f_{nl} represents the *live fanout* of a non-leaf node. For example, in node *B* of Figure 10, there are 3 entries alive at x=5, and 4 at x=15. So the live fanouts of node B are 3 and 4 respectively at these two *x*-coordinates. Note that leaf and non-leaf nodes are distinguished because their structural changes are different.

The live nodes at some *x*- coordinate increase 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

[1] The space consumption of aP-trees is higher than that of MVB-trees ($O(n/b)$) due to the 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.

non-leaf levels, key splits will happen only when the number of entries in the new node exceeds the strong version overflow threshold. To distinguish this, we define *split points* as follows. The *split point* SP_l of a leaf node denotes the number of entries in a leaf node before it is key split. Similarly SP_{nl} corresponds to the *split point* of a non-leaf node. Let b_l and b_{nl} represent the block capacities of leaf and non-leaf nodes respectively. Then we have:

$$SP_l = b_l \text{ and } SP_{nl} = b_{nl} \cdot P_{SVO} \quad (4.1)$$

As shown in [Yao78], 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 as follows:

$$f_l = SP_l \cdot ln2 \text{ and } f_{nl} = SP_{nl} \cdot ln2 \quad (4.2)$$

An aP-tree consists of multiple logical B-trees, where more recent trees have larger heights as more insertions are performed. The height h of the last logical B-tree is:

$$h = 2 + \left\lceil \log_{f_{nl}} \frac{n / f_l}{SP_{nl}} \right\rceil \quad (4.3)$$

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

$$size(aP) = \sum_{i=0}^{h-1} N_i \quad (4.4)$$

The estimation for N_0, the total number of leaf nodes, is relatively straightforward, 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 (i) increases the total number of nodes by 2, and (ii) the number of live nodes by 1. 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:

$$N_0 = 2V_l + 1 = 2n / f_l - 1 \quad (4.5)$$

A similar analysis, however, does not apply to the estimation for N_i of non-leaf 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.

$$L_1 = SP_l$$

$$L_i = f_l \cdot f_{nl}^{i-2} \cdot SP_{nl} \quad (1 < i \leq h-1) \quad (4.6)$$

where SP_l, SP_{nl} and f_l, f_{nl} are split points and live fanouts for leaf and non-leaf nodes respectively as defined earlier. 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 entry insertions at each level is also n. Notice, however, that this estimation excludes strong version overflows because: (i) the number of strong version overflows is considerably lower than n; so omitting it will not bias the results significantly, and (ii) although capturing strong version overflows is straightforward, it would lead to excessively complicated equations.

Recall that a node already contains a number of entries (version copied from the previous node) when it is created. Another observation is that this number equals the number of live entries in the previous node. Since the average live fanout of non-leaf nodes is f_{nl}, it follows that a node contains f_{nl} initial entries on average. Therefore, a

node will, on the average, take (b_{nl}–f_{nl}) entries before it dies. Note that, 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 $(n–L_2)/(b_{nl}–f_{nl})$.

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_1:

$$N_1 = [\sum_{j=2}^{SP_{nl}} \frac{(L_2 - L_1)/(SP_{nl} - 2)}{(b_{nl} - j)}] + \frac{n - L_2}{b_{nl} - f_{nl}} \qquad (4.7)$$

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

$$N_i = [\sum_{j=2}^{SP_{nl}} \frac{(L_{i+1} - L_i)/(SP_{nl} - 2)}{(b_{nl} - j)}] + \frac{n - L_{i+1}}{b_{nl} - f_{nl}} \qquad (1 \le i \le h\text{-}2) \qquad (4.8)$$

Now it remains to clarify the estimation of N_{h-1}, which is different from the other non-leaf levels on two aspects: (i) there is no LuP for the higher level; (ii) 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, we have:

$$N_{h-1} = \sum_{j=2}^{\lceil n/(f_l \cdot f_{nl}^{h-2}) \rceil} \frac{(n - L_{h-1})/(\lceil n/(f_l \cdot f_{nl}^{h-2}) \rceil - 2)}{(b_{nl} - j)} \qquad (4.9)$$

Replacing variables in equation (4.4) correspondingly with results in equations (4.1 to 4.9), we obtain the cost model that predicts the structure size of the aP-tree. Note that this estimation does not include the size of the root table, which stores one entry for each root node. As will be shown in the experimental section, however, the size of the root table is negligible. Furthermore in this paper we assume that each disk page corresponds to one structure node; hence the model also gives the number of disk 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 2 paths from the root to the leaf level of a B-tree. Since the 2 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:

$$NA(VRA) = 2h - 1 = 3 + 2\left\lceil \log_{f_{nl}} \frac{n/f_l}{SP_{nl}} \right\rceil \qquad (4.10)$$

Thus the cost of answering a WA query, which involves two VRA queries, is given by equation 4.11. Notice that the query costs involve a very low constant value irrespective of the sizes and positions of the queries.

$$NA(WA) = 2NA(VRA) = 6 + 4\left\lceil \log_{f_{nl}} \frac{n/f_l}{SP_{nl}} \right\rceil \qquad (4.11)$$

5. Experiments

In this section, we evaluate the sizes and query performance of aP- and aR-trees with synthetic and real datasets. All queries are quadratic, i.e., both sides of each query window have the same length, which is represented as a percentage of the unit axis. In our implementation, we optimize the performance of both structures by storing only necessary information in leaf and non-leaf entries. For example, for aR-trees, points are stored in leaf entries while MBRs are stored in non-leaf entries. For aP-trees, on the other hand, the x_{end} field of each entry does not need to be stored (see section 3.1). The page size is set to 4,096 bytes, for which the leaf and non-leaf node capacities of aP-trees are 255 and 204 respectively, while the corresponding figures for aR-trees are 255 and 170 (more information is needed in an intermediate aR-entry to store its MBR). The P_{svo} parameter of the aP-tree is set to 0.5 in all cases.

5.1 Uniform datasets

In the first set of experiments, datasets are generated uniformly in a unit square universe, ensuring that no two points have the same *x*-coordinate. Query performance is measured by the average number of node accesses in answering a workload consisting of 500 queries. Each query in the same workload has the same side length, ranging from 10% to 60% of the universe axis, resulting in query areas from 1% to 36%. The position of a query is randomly generated in the universe. Figure 11a demonstrates the number of node accesses as a function of the query side for the aP- and aR-tree indexing a uniform dataset with 150K points. It is clear that the aR-tree is comparable to the aP-tree only for very small query windows, and its performance keeps increasing linearly with the query side. On the other hand, the performance of the aP-tree stabilizes around 10 accesses irrespective of the query side, which makes it considerably more efficient than its competitor. Further, since the height of the aP-tree is 3, the cost (10 accesses) is exactly our estimation given by equation (4.11).

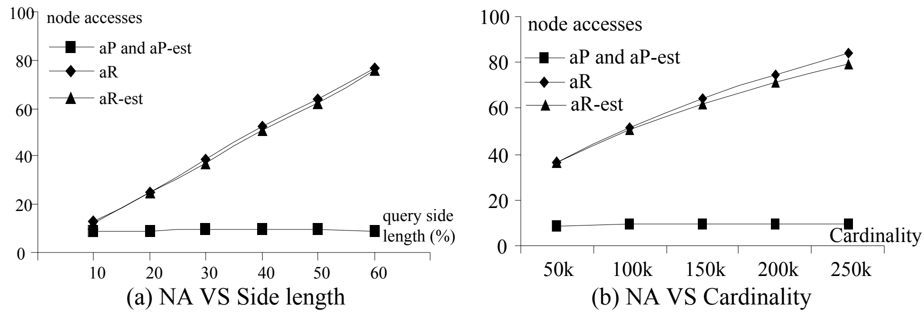


Fig. 11. Query costs of WA queries

Figure 11b demonstrates the query cost (using the workload with side length 50%) as a function of the cardinality for uniform datasets with 50K, 100K, 150K, 200K, and 250K points. The performance of the aR-tree deteriorates quickly when the cardinality increases, while that of the aP-tree remains constant (at 10 node accesses), as there is

no change in the height. Notice that our estimation (by equation 2.1) of the aR-tree performance is very accurate in all cases, producing error less than 5%.

As discussed in the analysis, the excellent performance of aP-trees comes at the expense of extra space consumption. Figure 12 shows the sizes of aP- and aR-trees as a function of the data cardinality. aP-trees consume more space than the corresponding aR-trees, because each insertion must create a new entry in each node on the insertion path, while, for aR-trees, new non-leaf entries are needed only when new nodes at the lower level are spawned. Furthermore, despite the fact that the size complexity of the aP-tree is $O(n/b \log_b n)$, its growth is quite linear with the cardinality. This is expected because the factor $\log_b n$ in the complexity actually corresponds to the height of the tree. Therefore, the size of the aP-tree grows linearly as long as its height remains unchanged, which is true for the 5 cardinality values in the figure. Note that the estimated values (by equation 4.4) capture the actual behavior of the aP-tree very well, producing error below 5% in all cases. It is worth mentioning that the sizes of the root tables of the aP-trees are about 0.1% of the total tree sizes.

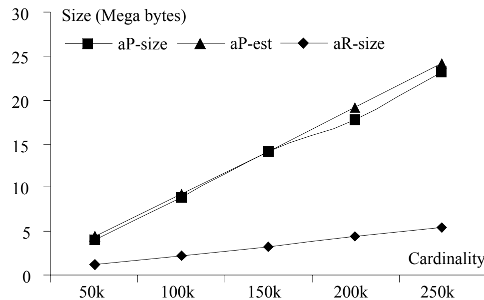


Fig. 12. Size comparison of aP- and aR-trees

5.2 Non-uniform datasets

In this section we compare the two structures using 6 non-uniform datasets described as follows: (i) *gauss* contains 100K points distributed following the gaussian distribution; (ii) *skewed* has the same cardinality but the distribution is skewed (*gauss* and *skewed* were generated using the GSTD utility [TSN99]); (iii) *CFD1* (52K points) and *CFD* (200K points) are vertex data from various Computation Fluid Dynamic models measured for a cross section of a Boeing 737 wing with flaps out in landing configuration [Web1]; (iv) *SCG* contains 62K points representing gravity data for South California [Web2]; (v) *SCP* contains 46K points describing places in South California [Web3]. The datasets are shown in Figure 13.

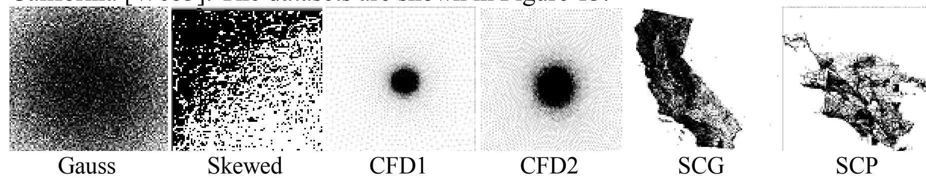


Fig. 13. Visualization of non-uniform datasets

To study query performance, we generated workloads specifically for each dataset such that the queries in a workload distribute similarly to the corresponding dataset in order to avoid queries falling into empty areas. Figure 14 illustrates, for each dataset, the number of node accesses as a function of the query length. It is clear that aP-trees outperform aR-trees in all cases, and the difference is up to an order of magnitude. An interesting observation is that, when data is skewed, the processing cost of aR-trees usually drops when the query length exceeds a threshold. This happens because skewed data lead to skewed MBRs of the nodes. Hence when the query rectangle is large enough, many MBRs tend to assemble inside the query window, so fewer MBRs intersect the window sides. Note that this phenomenon does not exist for uniform datasets, where the MBRs of nodes are also uniformly distributed.

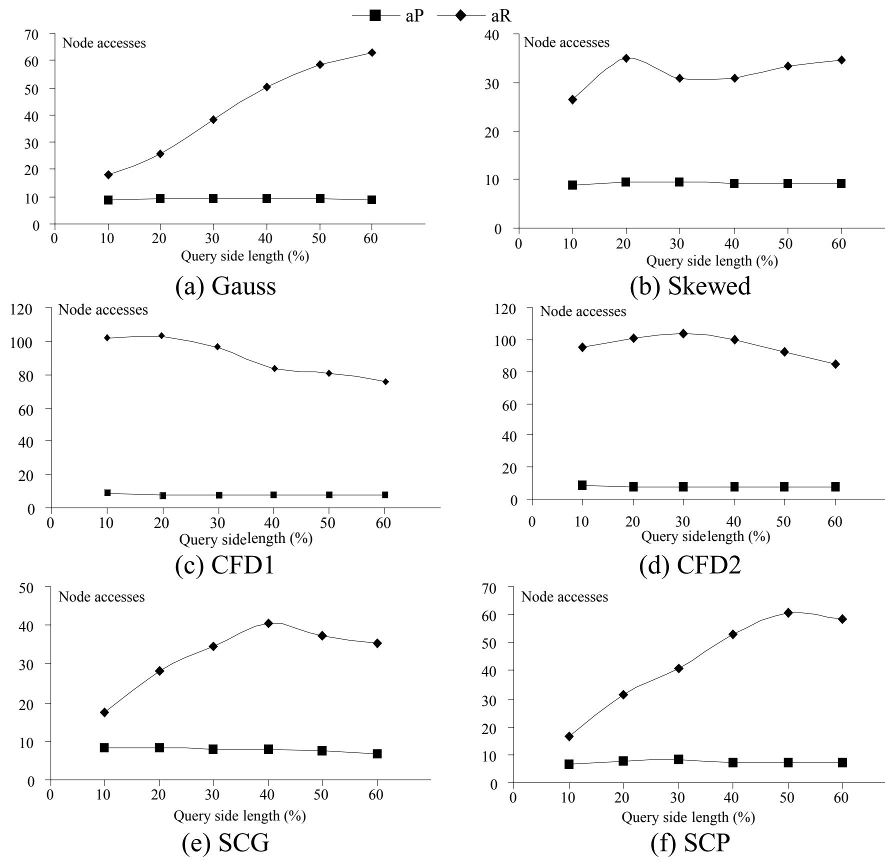


Fig. 14. Query performance for non-uniform datasets

Finally, Figure 15 demonstrates the sizes of corresponding aP- and aR-trees for the datasets above, as well as the estimated values from our model (equation 4.4). The difference between the two structures is similar to that of the uniform case, and our cost models again predict the performance very accurately.

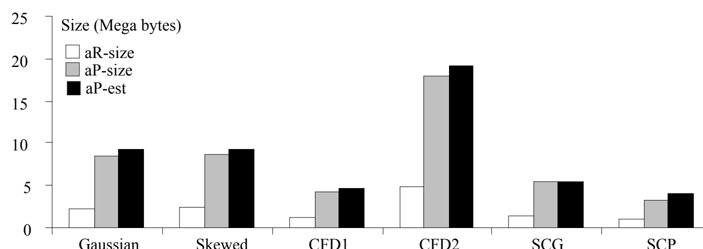


Fig. 15. Sizes of both structures for non-uniform datasets

6. Conclusions and Future Work

This paper addresses the problem of processing WA queries on planar points, proposing a new indexing structure, the aP-tree, that answers any WA query in logarithmic worst case time. Extensive experiments have shown that the aP-tree outperforms its most significant competitor, the aR-tree, by up to an order of magnitude. Its main advantage compared to any R-tree based method, is that the query cost is independent of the window size. We believe that, given the significant performance benefits and the dropping cost of secondary memory, the extra space requirements of the aP-tree do not constitute a serious shortcoming, especially for applications that require fast, real-time query processing. In addition, we present efficient cost models (with less than 5% error) that predict the structure size of the aP-tree and, consequently query performance.

An interesting direction for future work is to investigate whether it is possible to improve the space requirements of the aP-tree, or to prove a theoretical bound for any indexing structure that answers WA queries in logarithmic worst case time. A challenging extension is to apply the same approach to support more general spatial objects (such as rectangles, spheres, etc), possibly in multiple dimensions.

Acknowledgments

This work was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

References

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

[BKS+90] Beckmann, B., Kriegel, H.P., Schneider, R, Seeger, B. The R*-tree: An Efficient and Robust Access Method. *ACM SIGMOD*, 1990.

[BS96] Bercken van den, J., Seeger, B. Query Processing Techniques for Multiversion Access Methods. *VLDB*, 1996.

[JL98] Jurgens, M., Lenz, H. The Ra*-tree: An Improved R-tree with Materialized Data for Supporting Range Queries on OLAP-Data. *DEXA Workshop*, 1998.

[KF93] Kamel, I., Faloutsos, C. On Packing R-trees. *CIKM*, 1993.

[KGT+] Kollios, G., Gunopulos, D., Tsotras, V., Dellis, A., Hadjieleftheriou, M. Indexing Animated Objects Using Spatiotemporal Access Methods. To appear in *IEEE TKDE*.

[KKK99] Kim, J., Kang, S., Kim, M. Effective Temporal Aggregation using Point-based Trees. *DEXA*, 1999.

[KS95] Kline, N., Snodgrass, R. Computing Temporal Aggregates. *IEEE ICDE*, 1995.

[KTF98] Kumar, A, Tsotras, V., Faloutsos, C. Design Access Methods for Bitemporal Databases. *IEEE TKDE* 10(1): 1-20, 1998.

[LM01] Lazaridis, I., Mehrotra, S. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. *ACM SIGMOD*, 2001.

[PKZ+01] Papadias, D., Kalnis, P., Zhang, J., Tao, Y. Efficient OLAP Operations in Spatial Data Warehouses. *SSTD,* 2001.

[PT01] Pedersen, T., Tryfona, N. Pre-aggregation in Spatial Data Warehouses. *SSTD*, 2001.

[PTK+02] Papadias, D., Tao, Y., Kalnis, P., Zhang, J. Indexing Spatio-temporal Data Warehouses. *IEEE ICDE*, 2002.

[ST97] Salzberg, B., Tsotras, V. A Comparison of Access Methods for Temporal Data. *ACM Computing Surveys*, 31(2): 158-221, 1997.

[TP01] Tao, Y., Papadias, D. The MV3R-tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. *VLDB*, 2001.

[TPZ02] Tao, Y., Papadias, D., Zhang, J. Efficient Cost Models for Overlapping and Multi-Version Structures. *IEEE ICDE*, 2002.

[TS96] Theodoridis, Y., Sellis, T. A Model for the Prediction of R-tree Performance. *ACM PODS*, 1996.

[TSN99] Theodoridis, Y., Silva, J. Nascimento, M. On the Generation of Spatiotemporal Datasets. *SSD*, 1999.

[VV97] Varman, P., Verma, R. An Efficient Multiversion Access Structure. *IEEE TKDE*, Vol. 9, No. 3, pp. 391-409, 1997.

[Web1] http://www.cs.du.edu/~leut/MultiDimData.html

[Web2] http://www.gps.caltech.edu/~clay/gravity/gravity.html

[Web3] http://dias.cti.gr/~ytheod/research/datasets/spatial.html

[Yao78] Yao, A. Random 2-3 Trees. *Acta Informatica*, Vol. 2(9), 159-179, 1978.

[YW01] Yang, J., Widom, J. Incremental Computation and Maintenance of Temporal Aggregates. *IEEE ICDE*, 2001.

[ZMT+01] Zhang, D., Markowetz, A., Tsotras, V., Gunopulos, D., Seeger, B. Efficient Computation of Temporal Aggregates with Range Predicates. *ACM PODS*, 2001.

[ZTS02] Zhang, D., Tsotras, V., Seeger, B. Efficient Temporal Join Processing Using Indices. *IEEE ICDE*, 2002.