Efficient Historical R-trees

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

Abstract

The Historical R-tree is a spatio-temporal access method aimed at the retrieval of window queries in the past. The concept behind the method is to keep an R-tree for each timestamp in history, but allow consecutive trees to share branches when the underlying objects do not change. New branches are only created to accommodate updates from the previous timestamp. Although existing implementations of HR-trees process timestamp (window) queries very efficiently, they are hardly applicable in practice due to excessive space requirements and poor interval query performance. This paper addresses these problems by proposing the HR+-tree, which occupies a small fraction of the space required for the corresponding HR-tree (for typical conditions about 20%), while improving interval query performance several times. Our claims are supported by extensive experimental evaluation.

1. Introduction

The most fundamental type of query in spatial databases is the *window query*, which retrieves all objects that intersect a window specified by the user. In spatio-temporal databases, due to the inclusion of temporal information, there exist two types of window queries: (i) *timestamp* (or *timeslice*) queries that retrieve all objects that intersect a window at a specific timestamp, and (ii) *interval* queries, which involve several continuous timestamps. Since window queries, especially timestamp queries, are usually the building blocks for other more sophisticated operations, their efficient processing is vital to the overall system performance. Supporting such queries in spatio-temporal databases demands new querying languages, modeling methods, novel attribute representations [5], and, very importantly, new access methods [14].

Considerable work has been done on indexing static spatial objects [6]. Probably the most popular index is the R-tree [7], a balanced structure that clusters objects by their spatial proximity. R-tree variants are currently incorporated in many commercial DBMS. A straightforward solution towards indexing spatio-temporal data is to create an R-tree for each timestamp in history. Such an approach will certainly achieve excellent performance for timestamp queries as they degenerate into traditional window queries. However, an obvious disadvantage would be the excessive space required to store all the trees. In fact, one may notice that it is not necessary to preserve a complete tree on each timestamp due to the fact that consecutive trees may have a lot of identical branches. This is especially true, if only a small percentage of the objects move at each timestamp.

The MR-tree [16] is the first structure that takes advantage of this observation. In MR-trees, consecutive trees share branches when the underlying objects do not move and new branches are only created to accommodate changes from the previous timestamp. The first concrete update algorithms were presented in [9], which proposed the HR-tree based on the same idea. No experimental evaluation was available for these methods until [10] compared the HR-tree with some 3D R-tree implementations (in 3D R-trees time is incorporated as an extra dimension). It was revealed that, HR-trees outperform 3D R-trees on timestamp and short-interval queries. This is due to the fact that timestamp query performance of 3D R-trees does not depend on the live entries at the query timestamp, but on the total number of entries in history. Since all objects are indexed by a single tree, the size and height of the tree is expected to be larger than that of the corresponding HR-tree at the query timestamp. However, the space requirements of HR-trees are still prohibitive in practice, because for most typical datasets HR-trees almost degenerate to independent R-trees, one for each timestamp. Furthermore, their performance deteriorates very fast for interval queries as the interval length increases. Thus, a space-efficient method that performs satisfactorily on both timestamp and interval queries is necessary, given the fundamental importance of these queries in any system that deals with historical information retrieval.

In this paper we propose the HR+-tree, which outperforms the HR-tree significantly with respect to both space requirements and query performance. To be specific, the new method consumes *less than 20% of the space* required by HR-trees yet *answers interval queries several times faster*. Meanwhile, it *processes timestamp queries as efficiently* as HR-trees. The rest of the paper is organized as follows. Section 2 introduces the HR-tree, discusses its advantages, and analyzes its problems. Section 3 presents HR+-trees and the corresponding update and query processing algorithms. Section 4 contains an extensive experimental evaluation, while section 5 summarizes the contributions and provides directions for future work.

2. Historical R-trees

Historical R-trees (HR-trees) [9] are based on the overlapping technique [4, 12] that transforms a single version data structure into a partially persistent one. The structure maintains an R-tree[1] for each timestamp, but common branches of consecutive trees are stored only once in order to save space. Figure 1 illustrates part of an HR-tree for timestamps 0 and 1. Node A_0 is shared by both trees meaning that its content has not changed during these timestamps. Insertion is carried out as follows. First the leaf to insert the entry is found by applying the R*-tree [2] *choose subtree* algorithm. If the leaf node is shared by some earlier tree, it is duplicated, the entry is inserted in the new copy, and the update is propagated up to the root of the current tree, duplicating the internal nodes if they are shared by some earlier tree. Notice that the trees of previous timestamps are never modified.

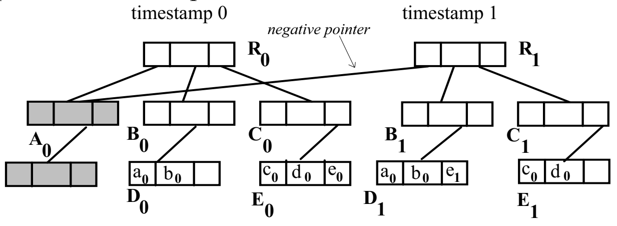


Figure 1: Example of HR-tree

A timestamp query is directed to the corresponding R-tree and search is performed inside this tree only. Thus, the query degenerates into a traditional window query and is handled very efficiently. HR-trees' excellent timestamp query performance, however, does not come for free. Although it is claimed in [10] that the structure can achieve up to 33% space savings with respect to the naïve multiple tree implementation, the size is still prohibitive for practical applications. Consider for instance a node capacity of 100 rectangles (a rather common value). Even, if less than 1% of the objects move between two consecutive timestamps, it is possible that the whole R-tree needs to be replicated since a moving object may cause the duplication of multiple nodes.

In the example of Figure 1, if the new version e_1 of e_0, is inserted in node D_0, it will cause the creation of two new nodes: D_1, which contains the entries of D_0 plus e_1, and E_1, which contains the entries of E_0 after the deletion of e_0. The change(s) should be propagated to the root (causing the creation of B_1 and C_1) so even if only one object changes its position, the entire path may need to be duplicated. It is obvious from the above discussion that in most typical situations, the HR-tree will contain multiple copies of the same object at different timestamps although the object has not moved (e.g., a_0, b_0, c_0 and d_0 in Figure 1). We call this phenomenon *version redundancy*. Although to some extent version redundancy is unavoidable for maintaining satisfactory timestamp query performance, it is rather excessive in HR-trees (this is experimentally demonstrated in section 4).

An interval query involving several timestamps should search the corresponding trees of the timestamps involved. In order to avoid multiple visits to the same node via different roots, we use[2] positive and negative pointers to distinguish exclusive and shared nodes. In Figure 1, for instance, when R_1 is copied from R_0, its pointers to nodes A_0, B_0, C_0 are all negative. Then, as new nodes are created in the current tree (e.g., B_1), the negative pointers (e.g., to B_0) are replaced with positive ones (to B_1). A general interval query on HR-trees can now be answered as follows: the tree associated with the earliest timestamp is searched first and all (positive and negative) qualifying pointers are followed to the leaves. Next the trees associated with the other timestamps are searched in chronological order by following only positive pointers. Interval query performance, however, is seriously affected by version redundancy and the large size of the structure. In the next section, we propose a new access method that overcomes the disadvantages of HR-trees.

3. HR+-Trees

In order to guarantee good timestamp query performance, HR-trees do not allow entries of different versions (timestamps) to be placed in the same node. HR+-trees break this constraint. Part of an HR+-tree is

[1] All implementations in the paper are R*-trees [2] since they are considered the most efficient R-tree variant.

[2] Interval queries were not discussed in the original HR-tree papers [9, 10].

shown in Figure 2 (the node capacity is 10 for all the following examples). The subscript of an entry denotes its version. Note that the leaf node *C* contains entries of two versions (i.e. timestamp 0 and 1). Entries *u* and *v* are the two parent entries of *C* at timestamp 0 and 1 respectively (i.e. node *C* is shared by two trees). Although *u* and *v* both point to *C*, their minimum bounding rectangles (MBRs) are different. In particular, *u* encloses objects alive at timestamp 0 (i.e., a_0, b_0, c_0, d_0), and *v* contains objects alive at timestamp 1 (i.e., d_0, e_1, f_1, g_1). Entry d_0 is bounded by both entries, which indicates that d_0 does not change its position at timestamp 1. Entries a_0, b_0, and c_0, however, are "invisible" to *v* as they have been deleted at timestamp 1. In HR+-trees, new nodes are created *only when an overflow occurs*; thus, space is utilized better. Tree construction algorithms (to be described shortly) ensure that the timestamp query performance does not degenerate.

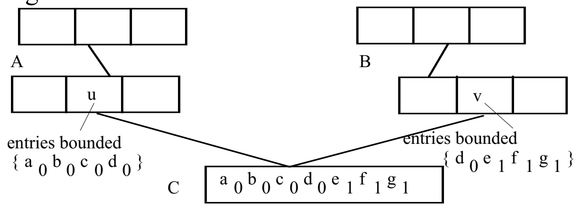


Figure 2: HR+-tree and HR-tree

The inclusion of different versions in the same node calls for some method to distinguish these versions. For example, in a query raised at timestamp 1, we have to separate a_0, b_0, and c_0 from the rest of entries in node *C*. This makes it necessary to store temporal information along with the entries (this is not needed in HR-trees). In our implementation, each entry has the form <*S*, t_{start}, t_{end}, *pointer*>. *S* denotes the MBR as defined in R-trees, while t_{start} (t_{end}) represents the timestamp that the corresponding entry is inserted (deleted). The lifespan of an entry is the semi-closed interval [t_{start}, t_{end}). If an entry has not been deleted until the current time (i.e. a currently *live* entry), its t_{end} is marked as "*" (a reserved word which means *now-time*).

The temporal information stored with each entry in the HR+-tree lowers the node capacity, which harms space utilization and query performance. Instead of storing the actual timestamps, which require 4 bytes (standard integer implementation) each, we keep the relative timestamps. The relative time is the actual time minus the creation time of the corresponding node. In this way, we use only 1 byte for each timestamp but for each node we need to store its creation time (4 bytes). Values of t_{start} and t_{end} are in the range [0, 255]; if this range is exceeded (e.g., entries have excessively long lifespans), appropriate data duplication is introduced to ensure correctness, which, however, is very infrequent in practice. With this mechanism, the fanout of nodes in HR+-trees is only about 8% smaller than that of HR-trees.

For leaf entries, the *pointer* points to the actual record; for intermediate entries, it points to a node at the next level. Figure 3 shows information for the corresponding entries in Figure 2 (the current timestamp is 1). Entries *d*, *e*, *f*, *g* are currently alive, while *a*, *b*, *c* are dead, i.e., they have been *logically deleted* at timestamp 1. An entry is physically deleted, only if its $t_{start} = t_{end}$.

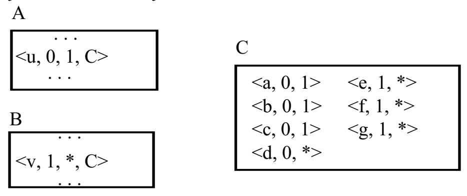


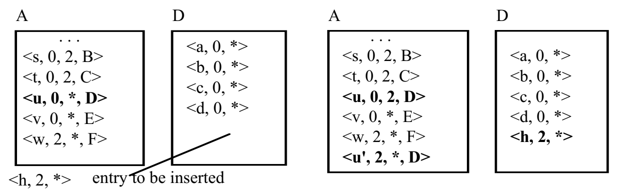
Figure 3: Entry information in the HR+-tree

In order to maintain good timestamp query performance, it is necessary to guarantee that each node contains a minimum number of entries alive at a given timestamp. Motivated by the multi-version B-tree [1], we ensure that each node in the HR+-tree satisfies the *weak version condition*, which states that, for all the nodes there must be either none or at least $B \cdot P_{weak}$ entries alive at any timestamp. *B* is the node capacity and P_{weak} is a tree parameter ranging from 0 to 1. This mechanism groups temporally adjacent entries together, so that during the processing of timestamp or short-interval queries only a small number of nodes needs to be accessed. To maintain the weak version condition, we applied the concept of *version split* in previous temporal access methods [8, 1, 15], as will be elaborated in subsequent sections.

3.1 Insertion algorithms and overflow handling

Like HR-trees, the HR+-tree is a *partially persistent* access method [12], in the sense that updates (insertions or deletions) can be applied to the current timestamp only. Similar to R*-trees, the insertion procedure of HR+-trees includes 3 main steps: (i) the leaf node to accommodate the new entry, is first located by the *choose subtree* algorithm; (ii) if entering the new entry causes the node to overflow, the *treat overflow* function is called; (iii) the information along the insertion path is adjusted correspondingly to reflect the changes. In the sequel we describe these algorithms in detail and elaborate their differences with those of R-trees.

3.1.1 The *choose subtree* algorithm. The *choose subtree* algorithm determines a leaf node to enter the new entry. To ensure the weak version condition, the leaf node should be *alive*, in the sense that it should contain some live entries. In Figure 4(a), during the insertion of entry *h* at timestamp 2, dead branches *s* and *t* are eliminated immediately. Among the remaining entries, the one to be followed is determined similarly to R*-trees: (i) if it is at the level just above the leaves, the one that incurs the *minimum overlap enlargement* is selected; (ii) if it is at a higher level, we select the one leading to the *minimum area enlargement* (ties are resolved as described in [2]).



(a) Before insertion (b) After insertion

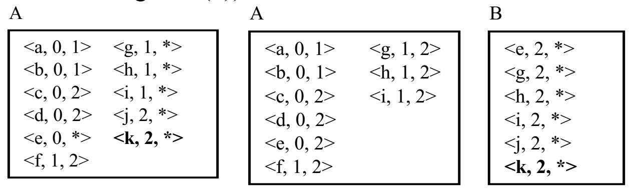
Figure 4: Duplicating an intermediate entry

Unlike conventional R-trees, however, the selected entry may be duplicated to guarantee good timestamp query performance. In Figure 4(a), for example, the leaf node selected by the *choose subtree* algorithm is *D*, which is reached by following the entry *u* in node *A*. If the insertion of <*h*,2,*> into *D* does not enlarge *D*'s spatial extent, the insertion does not incur any structural changes in the tree. If, however, the insertion causes enlargement, a new entry *u'* is created while *u* is logically deleted (its t_{end} is changed to 2). Entry *u* spatially bounds the entries (*a,b,c,d*) alive in the interval [0, 2), while *u'* bounds entries alive at timestamp 2. Figure 5 describes the *choose subtree* algorithm formally.

```
Algorithm Choose Subtree(new_entry)
1.  let N be the root associated with the current timestamp
2.  if N is a leaf, return N
3.  let S = {all the live entries in N}
4.  if N is just above the leaf level
5.      select an entry e from S such that inserting new_entry to e
        incurs the minimal overlap enlargement
6.  else (N is at a higher level)
7.      select an entry e from S such that minimum area
        enlargement is necessary
8.  if $e.t_{start}$ < current timestamp and e.MBR has changed
9.      insert a copy e' of e into N and set $e'.t_{start}$ to the current
        time
10.     set $e.t_{end}$ to the current time (delete e)
11. set N to the child pointed to by e (or e', if created) and go to
    line 2
```

Figure 5: Algorithm *choose subtree*

3.1.2 Handling overflows. Overflow occurs when an entry is inserted into a node that already contains the maximum number of entries. In this case, conventional R-trees split the corresponding node into two, optimising criteria such as the minimum overlap, area and margin. This type of time-independent split is called the *key split* in HR+-trees, and is performed only when all the entries in the node were inserted at the current timestamp. In Figure 6(a), node *A*, which generates an overflow due to the insertion of <*k*, 2, *> at timestamp 2, cannot be key split because there exist entries inserted at timestamp 0 and 1. In this case, a *version split* is performed instead to ensure the weak version condition. As shown in Figure 6(b), a version split generates a new node *B* containing all the live entries in node *A* with the following modifications: (i) all the entries in *B*, since they are inserted at the current timestamp, have t_{start} = 2; (ii) the entries that used to be alive in node *A* are logically deleted (t_{end} = 2); (iii) those entries in node *A* inserted at the current time (e.g., *j, k*) are *physically* deleted from *A*. Notice that overflow cannot persist in the new node generated from a version split (e.g., node *B* in Figure 6(b)).



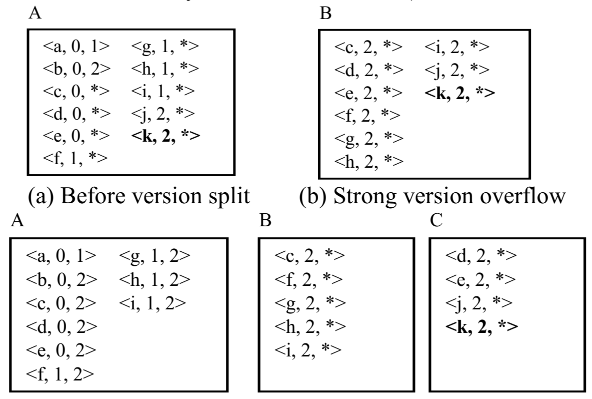
(a) Before version split (b) After version split

Figure 6: Example of version split

Version splits result in *version redundancy*. An object that remains static at a certain position for a number of timestamps should ideally be represented by a single record. HR+-trees, like most transaction time access methods, permit some redundancy in order to achieve query efficiency. In the above example, entries *e,g,h* and *i* are duplicated in both the new and the old node although the objects remain static. This is necessary because, without version redundancy, an object that is clustered well with other objects at some timestamp may not necessarily be clustered well at other timestamps if it is placed in the same node. In addition to space overhead, version redundancy complicates query processing since it may cause duplicate visits to the same node (for intermediate level copies), or duplicate reports of the same result (for leaf level copies). We will discuss how to avoid these problems in section 3.3.

The new node created after a version split may be almost full, so that a few insertions in subsequent timestamps will cause it to (version) split again resulting in more redundancy. In order to avoid this situation, we introduce the *strong version overflow* that occurs if the new node contains more than $B \cdot P_{SVO}$ entries, where *B* denotes the node capacity and P_{SVO} is a tree parameter (this is also motivated by related work in temporal data structures [1]). Strong version overflows are treated by key splits. A key split just distributes the entries of a node into two new ones, optimising the spatial criteria of R* splits [2], and does not incur version redundancy. Notice that unlike R*-trees, overflows are always treated with splits (no re-insertions are attempted the first time a node overflows), because re-insertions can potentially lead to version splits in other nodes (the same approach is followed in our implementation of HR-trees).

In Figure 7 (P_{SVO} = 0.85), node *A* is version split into node *B*, which incurs a strong version overflow and is key split into nodes *B* and *C*. Note that, in Figure 7(c), entry modifications are made similarly to Figure 6(b). Corresponding entries are inserted into nodes at the higher levels to reflect the changes, and thus, the split may propagate up to the root. Figure 8 formally describes the *treat overflow* algorithm (algorithm *key split* is omitted because it is exactly the same as R-trees).



(a) Before version split (b) Strong version overflow

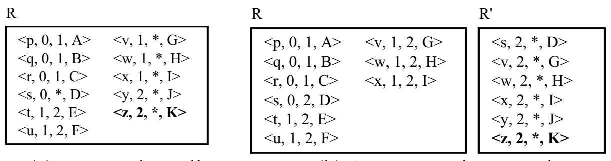
(c) Final situation after a key split

Figure 7: Example of strong version overflow

```
Algorithm Treat Overflow
1. set N to the node that incurs overflow
2. if all entries in N were inserted at the current timestamp
3.     key split N into itself and N_1 and go to line 13
4. create a new node N_1
5. for each live entry e in N
6.     duplicate e to e' and enter e' into N_1
7.     if e was inserted at the current timestamp
8.         physically delete e
9.     else
10.        modify the deletion time of e and the insertion time of e'
           to the current time
11. if N_1 strong version overflows
12.    key split it into N_1 and N_2
13. return N, N_1 (and N_2, if created)
```

Figure 8: Algorithm *treat overflow*

There can be multiple roots in an HR+-tree, and each root has a *jurisdiction interval,* which is the minimum bounding lifespan of all the entries in the root. The root of a new (logical) tree is created when the root of the previous tree incurs a version split. In Figure 9(b), *R'* is generated by the version split of root *R*, due to insertion of entry *z*. The jurisdiction interval of *R* contains timestamps 0 and 1, while *R'* becomes the root for the current timestamp. As with many *transaction time access methods*, a root table is maintained to record the corresponding roots for different timestamps. Obviously, the root table of HR+-trees has a smaller size than that of HR-trees since a root is now responsible for multiple timestamps. Figure 10 presents the insertion algorithm.



(a) Root to be split (b) A new root is created

Figure 9: Creating a new root

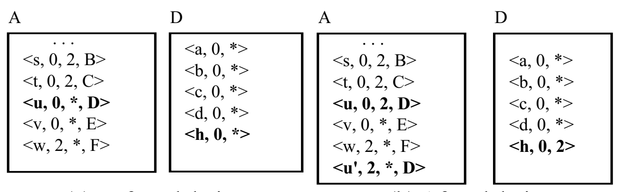
```
Algorithm Insert (new_entry)
1.  call choose subtree to locate the leaf node N_L to
    accommodate new_entry
2.  if N_L overflows after entering new_entry
3.      invoke treat overflow to handle the overflow
4.  ascend to the root and for each node N_I in the path
5.      adjust (insert, if necessary) appropriate entries to
        reflect the changes occurred in the levels below
6.      if N_I overflows invoke treat overflow
7.  if the root incurs a version split
8.      create a new entry in the root table for this
        timestamp
9.  if the root incurs a key split
10.     update the most recent entry in the root table
```

Figure 10: Algorithm *insert*

3.2 Deletion algorithms and underflow handling

Deletion in HR+-trees also follows the framework of R-trees: (i) algorithm *find leaf* identifies the leaf node containing the entry to be deleted; (ii) the entry is removed from the leaf node and the entries along the deletion path are adjusted; (iii) underflows are handled whenever necessary. Next we describe the deletion algorithms in detail.

3.2.1 The *find leaf* algorithm. Similar to insertions, deletions are allowed for the current timestamp; thus only live entries can be deleted. In order to locate the node containing the target entry, search is directed to the current tree. Then, the branch to be followed at each level contains both the spatial extents and the lifespan of the requested entry. Similar to insertion, we may need to create new entries to ensure good query performance. In Figure 11, for example, we want to delete <*h*, 0, *> from node *D*, which is reached by following entry *u* in node *A*. If the removal of *h* causes the MBR of node D to decrease, a new entry *u'* is created to bound the entries alive at the current timestamp (all entries but *h*) and *u* is logically deleted. In this way entries at every timestamp are bounded by the tightest MBR. Thus, deletion can actually cause overflows (e.g., due to insertion of *u'* in node *A*), which are handled as described in the previous section. Figure 12 presents the formal description.



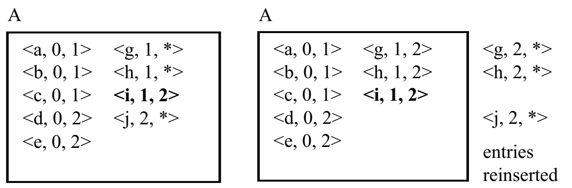
(a) Before deletion (b) After deletion

Figure 11: Duplicating an entry during deletion

```
Algorithm Find Leaf (entry_to_find)
1. let N be the node being considered
2. if N is a leaf
3.   if N contains entry_to_find return N
4.   else return not found
5. for each live entry e in N
6.   if e.MBR contains entry_to_find.MBR and e.lifespan
     contains entry_to_find.lifespan
7.     call find leaf passing the node pointed by e
8.   if entry_to_find was found (return from recursion)
9.     if e was inserted earlier and e.MBR has changed
10.      insert a copy e' of e into N and set $e'.t_{start}$ to the current
         time
11.      set $e.t_{end}$ to the current timestamp
12.    return found
13. return not found
```

Figure 12: Algorithm *find leaf*

3.2.2 Handling underflows. Underflow occurs as the consequence of violation of the weak version condition after deletion. It is not hard to see that it is always the current timestamp that fails to have enough live entries when an underflow happens. In the sequel, we will describe three alternatives to handle underflows. The first approach, derived directly from R*-trees, reinserts the live entries in a node that underflows. Figure 14(a) demonstrates a leaf node *A* that generates an underflow after deleting the entry <*i*, 1, *> ($P_{weak} = 0.4$). The deletion of *i* will cause only three entries (*g*, *h*, and *j*) to remain alive at the current timestamp. Thus, the weak version condition is violated and *g*, *h*, and *j* are reinserted, while node *A* is modified as described in Figure 13(b). Note that all the entries reinserted have their t_{start} set to the current time, while their original entries are deleted from *A*. Entry *j* is physically deleted from node *A* because it was inserted at the current timestamp.

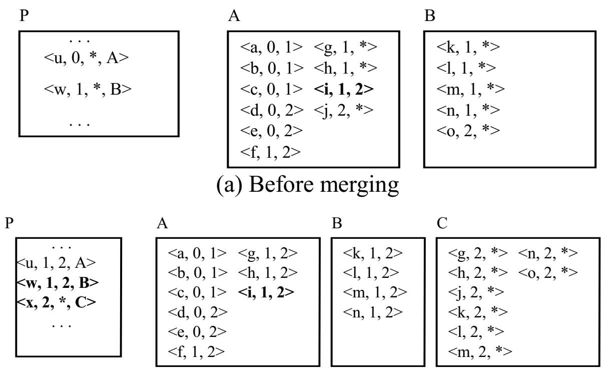


(a) Node underflow (b) Entries to be reinserted

Figure 13: Reinserting entries in a node

Entry re-insertion may lead to version redundancy and should be minimised. The second alternative is based on the observation that a node which underflows may have enough entries after subsequent insertions. In Figure 13(a), for example, node *A* may no longer underflow if some entry is inserted later at the same timestamp. Therefore, we do not handle the underflow immediately, but simply add it into a linked list storing all the nodes that incurred underflows at this timestamp. Before processing the first record of the next timestamp, we check each node in the linked list and reinsert its live entries if the underflow has not been recovered. Experimental results show that, for typical datasets, on the average about 20% of the underflows are repaired at the end of a timestamp; thus re-insertion (and version redundancy) is reduced.

The third approach does not apply reinsertion at all, but tries to merge the node that underflows with a sibling node, as in B+-trees. The sibling node to be merged must be a live (i.e. a node containing some live entries) child of the same father, and should minimize the enlarged area of the merged node.



(a) Before merging

(b) After merging

Figure 14: Use merging to handle underflows

```
Algorithm Treat Underflow (using the merging approach)
1. let N be the node that underflows and let P be its parent
2. S = {the child nodes (other than N) pointed to by the live
   entries in P}
3. find the node $N_s$ such that, among all the nodes in S,
   merging $N_s$ with N gives the minimum area enlargement
4. create a node $T_1$ containing the live entries of $N_s$ and N
5. set $t_{start}$ of the entries in $T_1$ to the current time
6. if $T_1$ strong version overflows then key split to $T_1$ and $T_2$
7. if all the entries in N were inserted at the current time
8.   discard N
9. else set $e.t_{end}$ of the live entries to the current time
10. goto to line 7 for $N_s$
11. create entries in P for $T_1$ (and $T_2$, if created) and modify
    (delete, if necessary) the entries pointing to N and $N_s$
```

Figure 15: Algorithm *treat underflow*

In Figure 14(a), node *A* generates an underflow (after the deletion of entry <*i*, 1, *> there are only three live entries) and node *B* is identified for merging with *A*. Merging is similar to performing version splits in nodes *A* and *B*, the difference being that the live entries of both nodes are inserted in a single new node *C*. Corresponding entries are modified or inserted in the parent node *P* to reflect the changes. Note that it is possible that node *C* will overflow, in which case a key split is performed. Figure 15 describes the *treat underflow* algorithm using the third approach (the formal descriptions about the first two approaches are omitted since they are relatively straightforward). The deletion algorithm is described in Figure 16.

```
Algorithm Delete (entry_to_del)
1. invoke find leaf to locate the node N_L that contains entry_to_del
2. delete entry_to_del from N_L and invoke treat underflow if N_L underflows
3. ascend to the root and for each node N_I in the path
4.   adjust (insert or delete, if necessary) the entries in N_I to reflect the changes in the lower level
5.   if N_I overflows invoke treat overflow
6.   if N_I is not the root and underflows, invoke treat underflow
7. if the root has only one entry but not a data page
8.   make the child of the entry the new root
9.   update the most recent entry in the root table
10. if the root incurs a version split
11.   insert a new entry in the root table
```

Figure 16: Algorithm *delete*

3.3 Window query processing

Query processing in HR+-trees is similar to that of HR-trees. For timestamp queries, search is directed to the root whose jurisdiction interval covers the timestamp. After that, search proceeds to the appropriate branches considering both spatial and the temporal extents (lifespan). The processing of interval queries is more complicated; since a node can be shared by multiple branches, it may be visited many times during an interval query[3]. In Figure 17, for example, it is unnecessary to follow entry *v* in node *B* if we have already visited node *C* via entry *u* in node *A*.

In HR+-trees, duplicate visits are caused by version splits and entry reinsertion at intermediate levels. In these cases, a new entry pointing to the same child node is created from an old one. One approach to avoid duplicate visits is to store in the new entry, the spatial extents of the old one. In Figure 17, for instance, if information about entry *u* is stored in *v*, we can decide whether to visit node *C* (from *v*) by checking if entry *u* intersects the query window. Storing such information, however, will significantly lower the fanout of the tree.

[3] The method proposed to solve this problem in MVB-trees [3] cannot be applied in our case.

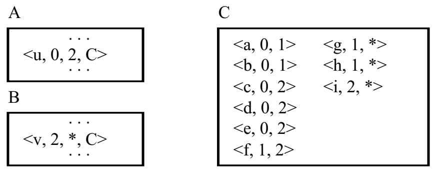


Figure 17: Duplicate visits in a query

Another solution is to perform (interval) queries in a breadth-first way. To be specific, we start with the set of roots whose associated logical trees will be accessed. By examining the entries in these nodes, we can decide the nodes that need to be visited at the next level. Instead of accessing these nodes immediately, we save their block addresses and check for duplicates. Only when we have finished all the nodes at this level, the nodes at the next level will be searched by the address information saved. It is evident that duplicate visits are trivially avoided.

Using this approach, an amount of memory is needed to maintain the block addresses. The memory overhead depends on the maximum number of nodes that will be accessed at a level in a query. Typically, this number is low and a very small fraction of buffer pages may be allocated for this purpose. We will show that such memory overhead hardly affects the performance in the experiment section. A priority heap is used to maintain the addresses in memory because such a heap can support search and update in logarithmic worst case time, though CPU time is negligible compared to I/O cost in most cases.

Similarly, redundant leaf entries, created by version splits, can be reported more than once in the result of interval queries. In order to perform duplicate elimination we distinguish the redundant versions by using negative ids (similar to the negative pointers of HR-trees). For example, in Figure 14, entry *g* can first be reported in node *A*, and then in node *C* (this is true for entries *h*, *k*, *l*, *m*, *n* as well). All copies have the same spatial extent, but different lifespans. When processing an interval query we only report the copy that contains the first timestamp of the interval. Continuing the example of Figure 14, assume an interval query covering timestamps 1-3. When entry *g* of node *C* is encountered, it will be discarded since it has a negative id and its lifespan starts at timestamp 2, implying that an earlier copy (that of node *A*) intersects the interval. Thus, duplicate elimination can be achieved without any additional space overhead.

4. Experiments

In this section, we compare HR+-trees with HR-trees through extensive experimentation. Due to the lack of real data, we generated synthetic datasets with real-world semantics using the GSTD method [13]. GSTD has been widely employed (e.g., [10, 11]) as a benchmarking environment for access methods dealing with moving points and regions. Each of the following datasets contains 10,000 regions with density 0.5 and is generated as follows. Objects' initial positions (i.e. at timestamp 0) are determined following a Gaussian distribution. In the subsequent 100 timestamps, objects move in way such that they eventually tend to scatter uniformly across the spatial universe, modeled as a unit square. Timestamps are modeled as float numbers ranging from 0 to 1 with granularity 0.01. At each timestamp, the percentage of objects that will change their positions is roughly the same and corresponds to the *agility* of the dataset, i.e., a dataset has agility *p*, if on the average *p%* of the objects change their positions at each timestamp. Unless specifically stated, the agility of the dataset in the sequel is 5%.

The performance of different access methods is measured by running *workloads*. Each workload contains 500 queries with the same (window) area and interval length (number of timestamps included in the interval). The areas of queries correspond to 1%, 5%, or 10% of the total universe. The intervals used involve 1 (timestamp queries), 5, 10, 15, or 20 timestamps, which account for up to 20% of the entire history. In the sequel, we refer to each workload as $\text{WRKLD}_{\text{area, length}}$ denoting workloads with different query areas and interval lengths. Queries are generated in a completely random manner: (i) the extents of each query window distribute uniformly in the spatial universe; (ii) the starting point of each query's interval distributes uniformly in the range [0, 1 – *length*]; (iii) unless specifically stated, the order of the queries is completely randomized.

The page size is set to 1,024 bytes for all the experiments and an LRU buffer with 200 pages (200K bytes) is assumed. For processing interval queries in HR+-trees, when additional memory is needed to maintain the block addresses, one memory page is allocated from the buffer. Conversely, when less memory is required for this purpose, the page is returned to the buffer. Each block address is represented by 4 bytes; thus for a query that needs to access 1000 leaf nodes (a really large query window), at most 5 pages are necessary at any time. Typical queries demand a very small amount of memory for maintaining the block addresses in memory.

There are two parameters for HR+-trees, namely, P_{weak} and P_{SVO}. Notice that P_{SVO} must be at least twice as large as P_{weak} to guarantee that the weak version condition can still hold after a key split due to a strong version overflow. Small values for P_{weak} and P_{SVO} reduce the number of underflows and version splits respectively, and hence avert version redundancy, leading to smaller tree size and better interval query performance. On the other hand, lowering P_{weak} reduces the node usage with respect to a single timestamp, while lowering P_{SVO} introduces more key splits; thus, timestamp query performance is compromised. A set of experiments was performed to explore the optimal settings for these parameters. The best overall performance was achieved for $P_{weak} = 0.4$ and $P_{SVO} = 0.85$ and in the sequel we use these values.

Four different HR+-tree versions were implemented. In particular, HR+_{NRML}, HR+_{DFR}, and HR+_{MRG} correspond to the versions with underflow treatments based on immediate reinsertion, deferred reinsertion, and merging respectively (as described in section 3.2.2). Interval queries are answered by searching the trees in the breadth-first manner. The last version, HR+_{PRE}, stores the extents of the previous version in each intermediate entry (as described in section 3.3) so that duplicate visits are avoided in interval queries without deploying the breadth-first search. Underflows in HR+_{PRE} are handled by reinserting the entries immediately. Assuming 1K page size, the node capacity for HR-, HR+_{NRML}, HR+_{DFR}, HR+_{MRG}, and HR+_{PRE} is 50, 46, 46, 46, and 33 respectively.

First, we compare the four HR+-tree implementations with respect to space requirements and query performance. Figure 18 shows the sizes of the HR+-trees for datasets with agility 5%, while Figures 19(a) and (b) illustrate the page accesses required for various interval lengths and window areas 1% and 10%, respectively. The diagrams suggest that HR+_{MRG} is less efficient than HR+_{NRML}, in terms of both space and query cost, indicating that reinsertion of entries is a better approach to handle underflows than merging. The HR+-_{PRE} variation performs noticeably worse than the other implementations; the speed-up achieved by storing the spatial extents of previous versions does not pay off due to lower node capacity. Furthermore, it is clear that HR+_{DFR} (deferred re-insertion) is the most efficient implementation; hence, we use this version to represent HR+-trees in the subsequent experimentation.

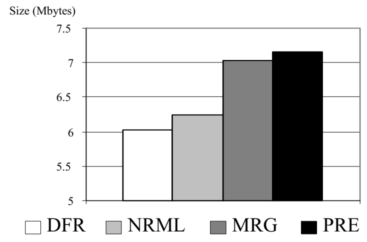
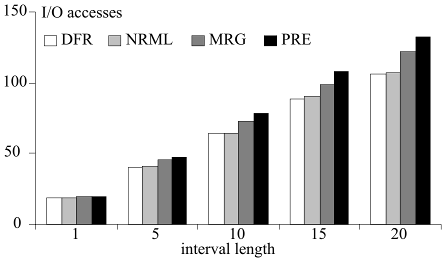
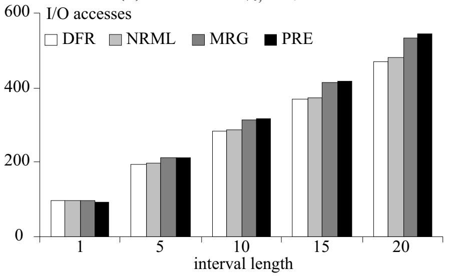


Figure 18: Size comparison of HR-tree variations

The remaining experiments compare HR-trees and HR+-trees on several aspects. Figure 20(b) shows the sizes of the two methods (in Megabytes) as a function of dataset agility. HR-trees grow very fast with agility and eventually the size of HR-trees appear to stabilize after 6% agility. At this point the size of HR-trees is around 33 Megabytes, 100 times the size when agility equals to 0% (static objects), implying that HR trees degenerate into individual R-trees, one for each timestamp. The sizes of HR+-trees, on the other hand, grow linearly with dataset agility at a reasonable speed. It is evident that HR+-trees are much smaller than HR-trees, though eventually (agility ⇓ 100%) both methods will tend to have similar sizes, as the high mobility of data forces both structures to degenerate into individual trees.



(a) WRKLD$_{1\%,\ 1\sim20}$



(b)WRKLD$_{10\%,\ 1\sim20}$

Figure 19: Performance of HR+-tree implementations

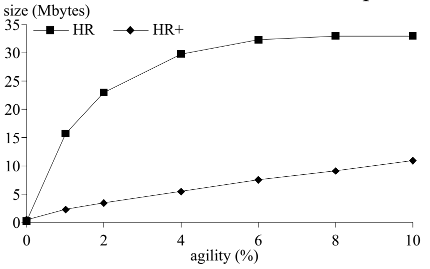
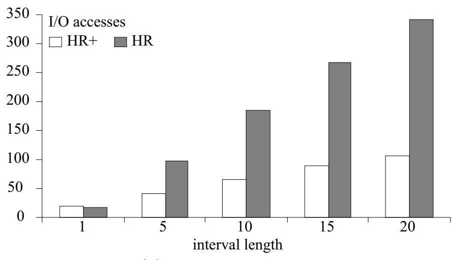


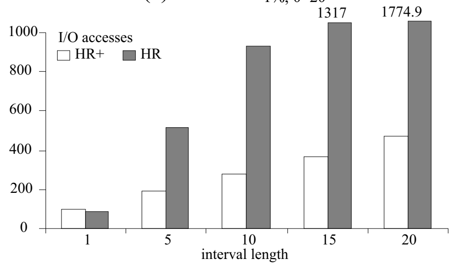
Figure 20: Size comparison under different agilities

Figure 21 compares performance using workloads of queries with windows covering 1% and 10% of the workspace. The page accesses are averaged over the number of queries in a workload, and shown as a function of the query length. For interval queries, HR+-trees outperform its competitor by a significant factor. The difference increases with the query length. For timestamp queries, HR+-trees are 5% to 10% less efficient than HR-trees. This is within our expectation due to the fact that node capacity is smaller for HR+-trees.

In some cases (e.g., batched workloads), queries can be sorted chronologically before being submitted to the system. This usually reduces the I/O accesses in the presence of buffers, due to the fact that queries with close temporal extents deploy similar parts of the index. In the following experiment, queries were sorted in each workload according to the starting time of their intervals. Figure 22 shows the page accesses of HR- and HR+-trees as a function of interval lengths for workloads involving 5%-area windows.



(a) WRKLD$_{1\%,\ 0\sim20}$



(b) WRKLD$_{10\%,\ 1\sim20}$

Figure 21: Query Performance (agility 5%)

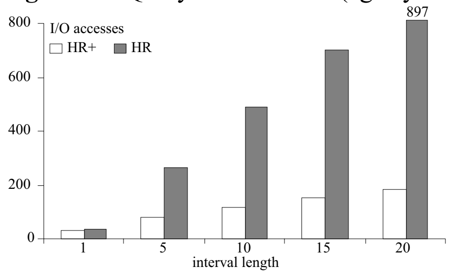


Figure 22: Sorted queries (WRKLD$_{5\%,\ 1\sim20}$)

Although the performance of both methods is improved, the HR+-tree receives larger improvements for all interval lengths. In particular, the HR+-tree outperforms the HR-tree even on timestamp queries. This is reasonable because each logical tree in HR+-trees is responsible for multiple timestamps; thus search may be performed in the same tree for adjacent timestamp queries, which utilizes the buffer more efficiently.

Finally, we investigated the performance of both methods when the cache size varies. We set the cache size to 100, 1000, 2000, 3000, 4000 pages (which accounts for 1.7% to 67.9% of the HR+-tree's size) and performed workloads with randomized timestamp queries. Figure 23 shows the results for queries involving windows covering 5% of the total workspace.

The HR+-tree performs significantly better than HR-trees when the buffer size increases. The efficiency of the HR+-tree eventually improves by more than 50%, whereas the HR-tree has only marginal improvements. For 1000 or more buffer pages, HR+-trees outperform HR-trees on all aspects (recall that the performance gap increases with the interval length). This is not surprising considering the large difference of the sizes of the two structures.

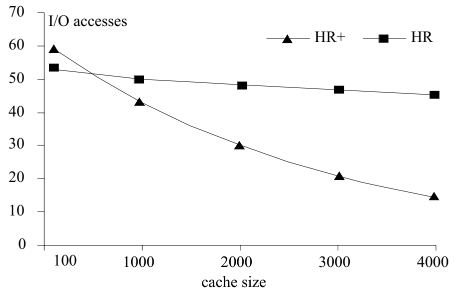


Figure 23: Random timestamp queries / buffer size

5. Conclusions and Future Work

In this paper, we propose HR+-trees a time and space efficient method for retrieval of historical information regarding moving regions and points. Compared to HR-trees, the most common method of handling timestamp queries, HR+-trees have the following properties:

∉ They consume a small fraction of the space required for the corresponding HR-trees (usually less than 20%).

∉ HR+-trees inherit HR-trees' efficiency on timestamp queries, but perform much better on interval queries.

∉ The improvement increases with the buffer size.

Furthermore, unlike 3D R-tree based methods [10], the HR+-tree does not assume that data are known a priori; thus can be used as an on-line spatio-temporal access method. Potential applications include urban planning and traffic management systems. Future investigation could focus on the following issues: (i) accurate analytical cost models for HR+-trees, (ii) query algorithms that can avoid all duplicate visits while incurring no memory overhead, and (iii) efficient algorithms of other operations (e.g. spatio-temporal joins) with HR+-trees.

Although spatio-temporal databases have received extensive attention during the past few years, many problems remain unsolved. Various applications place very different demands on the indexing methods. The previous structures, for example, may not be efficient for scenarios where region objects move at steady speeds. This is because, attempting to update the database whenever the objects change their positions will cause the STDBMS to spend most of time just handling the updates. Furthermore, this would result in huge space requirements. A better solution, though still not present, may be to store information about objects' motion patterns such as velocities. We are currently investigating solutions based on such ideas.

Acknowledgments

This work was supported from the Research Grants Council of the Hong Kong SAR, grants HKUST 6090/99E and HKUST 6070/00E.

References

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

[2] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD*, 1990.

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

[4] Driscoll, J., Sarnak, N., Sleator, D., Tarjan, R. Making Data Structures Persistent. *Journal of Computer and System Science* 38(1): 86-124, 1989.

[5] Forlizzi, L., Güting, R., Nardelli, E., Schneider, M. A Data Model and Data Structures for Moving Objects Databases. *ACM SIGMOD,* 2000.

[6] Gaede, V., Gunther, O. Multidimensional Access Methods. *Computer Surveys* 30(2): 170-231, 1998.

[7] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Aearching. *ACM SIGMOD*, 1984.

[8] Lomet, D., Salzberg, B. Access Methods for Multiversion Data. *ACM SIGMOD*, 1989.

[9] Nascimento, M., Silva, J. Towards Historical R-trees. *ACM SAC*, 1998.

[10] Nascimento, M., Silva, J., Theodoridis, Y. Evaluation of Access Structures for Discretely Moving Points. *International Workshop on Spatio-Temporal Database Management*, 1999.

[11] Pfoser, D., Jensen, C., Theodoridis, Y. Novel Approaches to the Indexing of Moving Object Trajectories. *VLDB*, 2000.

[12] Salzberg, B., Tsotras, V. A Comparison of Access Methods for Temporal Data. *ACM Computing Surveys* 31(2): 158-221, 1999.

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

[14] Theodoridis, Y., Sellis, T., Papadopoulos, A., Manolopoulos, Y. Specifications for Efficient Indexing in Spatiotemporal Databases. *SSDBM*, 1998.

[15] Varman, P., Verma, R. An Efficient Multiversion Access Structure, *IEEE TKDE* 9(3):391-409, 1997

[16] Xu, X., Han, J., Lu, W. RT-tree: An Improved R-tree Index Structures for Spatiotemporal Data. *SDH*, 1990.