# The MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries

Yufei Tao
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/~taoyf

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/~dimitris

## Abstract

Among the various types of spatio-temporal queries, the most common ones involve window queries in time. In particular, timestamp (or timeslice) queries retrieve all the objects that intersect a window at a specific timestamp. Interval queries include multiple consecutive timestamps. Although several indexes have been developed for either type, currently there does not exist a structure that can efficiently process both query types. This is a significant problem due to the fundamental importance of these queries in any spatio-temporal system that deals with historical information retrieval. Our paper addresses the problem by proposing the MV3R-tree, a structure that utilizes the concepts of multi-version B-trees and 3D R-trees. Extensive experimentation proves that MV3R-trees compare favorably with specialized structures aimed at timestamp and interval window queries, both in terms of time and space requirements.

## 1. Introduction

It is estimated that there will be more than 500 million mobile phone users by year 2002, and the figure is expected to surge to over 1 billion by year 2004. Spatio- -temporal database management systems (STDBMS) will play a crucial role in tracking users efficiently and providing better communication services. STDBMS are also important for *traffic supervision systems*, which monitor vehicle locations and motion patterns in order to provide services such as congestion prevention, route direction, speed limitation and so on. *Urban planning* is another domain where, rather than moving, objects (e.g., buildings) appear or disappear at certain time points. Related systems record the development of landscapes over the years, which makes it possible to retrieve urban situations at any given time in the past.

The above are only a few of the numerous applications in which STDBMS are essential. Traditional spatial database systems, however, focus on static objects. Supporting objects with dynamic behavior demands new querying languages, modelling methods, novel attribute representations [FGN+00], and, very importantly, specialized access methods [TSP+98]. Examples of such methods include the STR-trees and TB-trees [PJT00], which are aimed at efficient trajectory retrieval, and the TPR-trees [SJL+00] that focus on predicting objects' future locations by storing their current positions and velocities. The same problem was also addressed in [KGT99].

In this work we deal with retrieval of historical information, where the most common type of query processing involves window queries about objects that move (appear, disappear, change), usually in discrete time. *Timeslice* or *timestamp queries* retrieve all objects that intersect a window at a specific timestamp. *Interval queries* include several consecutive timestamps. Various indices have been proposed to support either type of queries. MR-trees [XHL90] and HR-trees [NS98] maintain a separate R-tree for each timestamp, but allow consecutive trees to share branches. The structures are very efficient for timestamp queries, as search degenerates into a static spatial window query for which R-trees are very efficient. Their disadvantage is extensive duplication of objects (even if they do not move) which leads to huge space requirements for most typical applications. As a side effect of this fact, their performance on interval queries is very poor.

Another technique is based on 3-dimensional R-trees where the third dimension corresponds to time. An object which does not change its position during a certain period of time is modelled as a 3D box, bounding both its spatial and temporal attributes. A moving object can be modelled by multiple boxes, each corresponding to a different version. The strength of 3D R-trees is that the temporal



**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

attribute is integrated tightly with the spatial attributes, so that interval queries can be answered efficiently. Another advantage is its economical space usage as redundant duplication is avoided. The most serious problem of this structure is its poor performance on timestamp queries. The query time no longer depends on the live entries at the query timestamp, but on the total number of entries in history.

In this paper, we attempt to overcome the shortcomings and combine the advantages of previous structures by proposing the MV3R-tree, an access method for retrieving the past locations of discretely moving objects. An MV3R-tree involves a multi-version R-tree (MVR-tree) and a small auxiliary 3D R-tree built on the leaves of the MVR-tree (i.e., not on the actual objects). MVR-trees involve several heuristics that take into account the features of R-trees to improve performance significantly. The space consumption of an MV3R-tree is up to an order of magnitude smaller than that of an HR-tree, while maintaining comparable timestamp query performance. Furthermore, the auxiliary 3D R-trees outperform traditional 3D R-trees for most queries. Our method constitutes a general approach for enhancing the performance of multi-version framework for multi-dimensional access methods.

The rest of the paper is organized as follows. Section 2 surveys the access methods directly related to our work, discusses their advantages, and analyses their problems. Section 3 presents MV3R-trees and the corresponding insertion, deletion and query processing algorithms. Section 4 contains an extensive experimental evaluation, while section 5 summarizes the contributions and provides directions for future work.

## 2. Related work

Since MVB-trees provide the initial motivation for the multi-version framework, section 2 starts with an overview of this structure. Next we describe HR-trees and 3D R-trees.

### 2.1 Multi-version B-trees

Multi-version B-trees (MVB-trees) [BGO+96] are extensions of B-trees that index the evolution of one-dimensional data in transaction time temporal databases [ST97], where insertions and deletions can only happen at the current time. Figure 2.1 illustrates a simple example. 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, the *pointer* points to a next level node. The temporal attributes $t_{start}$ and $t_{end}$ denote the time that the record was inserted and deleted in the database respectively. An entry is said to be *alive* at a timestamp $t$ if $t_{start} \leq t < t_{end}$, and *dead* otherwise (notice that the *lifespan* of an entry does not include $t_{end}$). The value of $t_{end}$ for currently live entries is "*", which denotes the special reserved word "NOWTIME". If a new entry is inserted at timestamp $t$, $t_{start}$ is set to $t$ and $t_{end}$ to *. On the other hand, if an entry is deleted, $t_{end}$ is changed (from *) to $t$. Deletions are logical; the actual records are not physically removed from the database.

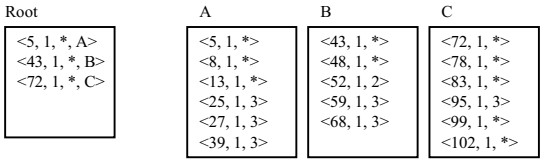



**Figure 2.1:** Example of MVB-tree

There can be multiple roots in an MVB-tree, and each root has a *jurisdiction interval,* which is the minimum bounding lifespan of all the entries in the root. Processing of timestamp (interval) queries starts by retrieving the corresponding root(s) whose jurisdiction interval(s) contains the queried timestamp(s). Then 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 and $b$ the node capacity (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 queries. Violations of this condition generate *weak version underflows*, which occur as a result of deletions at the current time.

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 current time. The value of $t_{end}$ of these entries in the original node is changed from * to the current time (in practice this step can be avoided since the deletion time is implied by the entry in the parent node). In Figure 2.2, the insertion of <28,4,*> at timestamp 4 (in the tree of Figure 2.1) 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.

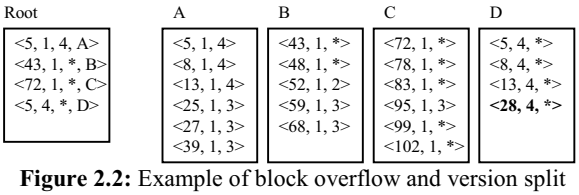



**Figure 2.2:** Example of block overflow and version split

Notice that, version splits create data redundancy for those entries duplicated (i.e., entries with keys 5, 8, and 13 in node D), as they should be ideally represented with one record (since they did not incur an update). Such

redundancy harms interval query performance as both the original and duplicated versions may need to be retrieved.

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 $[b \cdot P_{svu}, b \cdot P_{svo}]$ (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 2.1. 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 2.3).

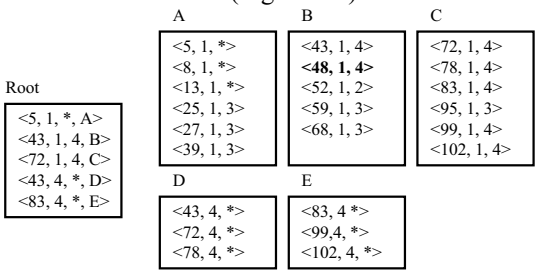



**Figure 2.3:** Example of weak version underflow

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 range query requires $O(\log_b M + r/b)$ I/O's, where *M* is the number of live objects at the queried timestamp, and *r* is the number of output objects. Both the space requirements and query performance are asymptotically optimal. A variation of MVB-trees which reduces the tree sizes by a constant factor can be found in [VV97]. Recently, the multi-version technique, has been applied to R-trees, to produce the BTR-tree [KTF98], a bitemporal access method, and the PPR-tree [KGT+01] for indexing multimedia objects that move continuously.

## 2.2 Historical R-trees

Historical R-trees (HR-trees) [NS98] are based on the overlapping technique [BKK+90], another framework for transforming a single version data structure into a transaction time access method. The structure maintains an R-tree for each timestamp, but common branches of consecutive trees are stored only once in order to save space. Figure 2.4 illustrates part of an HR-tree for timestamps 0 and 1. At timestamp 1, object e changes its position, so its old version $e_0$ should be deleted from the tree for timestamp 1, while its new version $e_1$ will be inserted. This causes the creation of two leaf 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) are 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. Trees of previous timestamps are never modified. Notice that node $A_0$ is shared by both trees, indicating that no object in its subtree issued any change at timestamp 1.

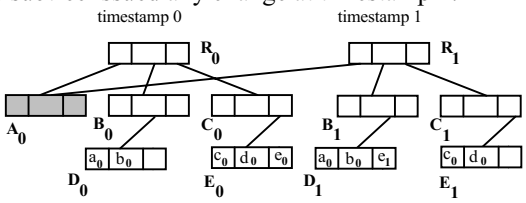



**Figure 2.4:** Example of an HR-tree

A timestamp query is directed to the corresponding R-tree and search is performed inside this tree only. Thus, the query degenerates into an ordinary window query and is handled very efficiently. An interval query should search the corresponding trees of all the timestamps involved. A query processing algorithm that uses "negative pointers" to avoid multiple visits to the same node via different parents was proposed in [TP01].

## 2.3 3D R-trees

The idea behind 3D R-trees is to view time as just another dimension and integrate it in the tree construction along with the other dimensions. The movements of 2D objects can be modelled as distinct boxes in three-dimensional space. The temporal projection denotes the period when the corresponding object remains static, while the spatial projections of the box correspond to the object's position and extents during the period. Whenever an object moves to another position, a new box is created to represent its new static period, position, and extents. Similarly, an interval query is also modelled as a box enclosing the spatial query window and query interval.

3D R-trees do not include a mechanism, such as the *weak version condition* in MVB-trees, to ensure that each node has a minimum number of live entries at a given timestamp. This affects performance of timestamp and short-interval queries, especially if dead space is taken

into account. Consider, for instance, the timestamp query in Figure 2.5. Since there is only one live entry at query timestamp *t*, the node has a lot of dead space with respect to *t*, meaning that there is a high chance that the query window intersects the bounding box but no object inside it. This problem is especially serious when there are many objects with long lifespans, as these objects will force the nodes that contain them to have long lifespans as well, leading to considerable mutual overlaps [KS91].

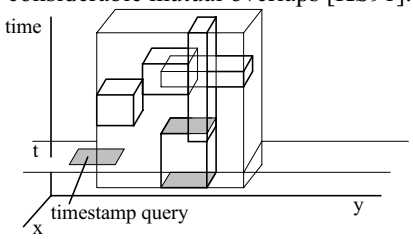



**Figure 2.5:** A timestamp query in 3D R-trees

Another reason for poor timestamp and short-interval query performance is due to the fact that, since there is a single tree for the whole history, the cost depends on the total number of records, rather than on the number of records alive at the queried timestamps, as in MVB- and HR-trees. On the other hand, long interval queries are efficient because: (i) there is no redundancy; (ii) R-trees, in general, optimize queries with similar extents along all dimensions (e.g., quadratic queries in 2D).

It is evident from the above discussion, that currently there does not exist a structure that can effectively handle both timestamp and interval queries. Towards this goal, in the next section, we propose the MV3R-tree, a structure originally motivated by MVB-trees, but with several heuristics to improve the overall performance significantly.

## 3. MV3R-Trees

A simple idea to deal with both timestamp and interval queries is to maintain two independent structures, an HR-tree and a 3D R-tree, and use the most appropriate one in each case. This, however, would require huge space (for the HR-tree). In addition, although this approach can deal effectively with timestamp and long interval queries, it is rather inefficient for short intervals, because neither 3D R-trees nor HR-trees can handle such intervals well. Multi-version 3D R-trees (MV3R-trees) overcome these problems by combining two structures: a multi-version R-tree (MVR-tree) and a small auxiliary 3D R-tree built on the leaf nodes of the MVR-tree. Figure 3.1 illustrates the general structure of the MV3R-tree.

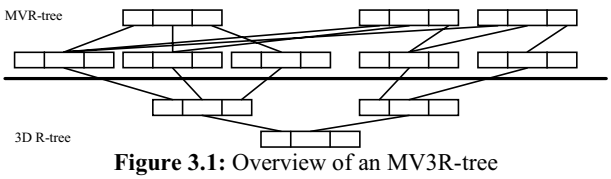



**Figure 3.1:** Overview of an MV3R-tree

As with MVB-trees, an MVR-tree can contain multiple R-trees, which we refer to as "logical trees". Each entry has the form <*S*, $t_{start}$, $t_{end}$, *pointer*>. *S* denotes the spatial minimum bounding rectangle (MBR) as defined in R-trees. The meanings of the other attributes are the same as in MVB-trees. MVR-trees inherit the concept of *weak version condition* from MVB-trees to guarantee that the number of live entries during a timestamp are either 0, or at least $b \cdot P_{version}$.

A small MVR-tree with height 2 is shown in Figure 3.2 ($b$=3, $P_{version}$=1/3). Object boxes (A to G) are sketched with thin lines, and leaf nodes (H, I, and J) with bold lines. K represents the root of the tree. Unbounded boxes (C, I, and K) are alive until the current timestamp. Objects are inserted in alphabetic order. At timestamp *t*, while objects A and B have already been deleted (C is alive), the insertion of D causes node H to overflow. A new node I is created to store a version duplicate of C, and D (notice that I and H store temporally adjacent parts of box C). The subsequent insertion of E and F will cause an overflow at node I which is handled by a key split that creates node J. Figure 3.2 shows the final situation after the insertion of G, and the deletion of D, E, G, F at subsequent timestamps.

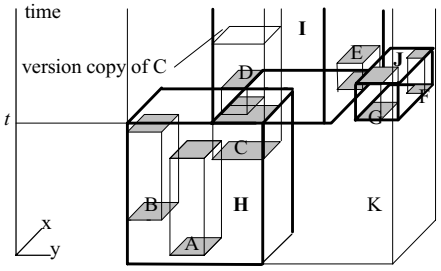



**Figure 3.2:** 3D visualization of a logical tree

Our implementation of MVR-trees involves several heuristics to reduce the structure size and improve query performance. Furthermore, these heuristics are applicable to other multi-dimensional access methods when they are converted to corresponding multi-version structures.

### 3.1 Insertion and overflow handling in MVR-trees

MVR-trees involve distinct insertion policies for leaf and intermediate nodes. The main difference is that for leaf nodes, we will try to avert version splits, which cause redundancy, as much as possible provided that the timestamp query performance is not compromised significantly. Since leaf nodes account for a large proportion of the structure's total size, avoiding redundancy will lead to significant reduction in the total space. Furthermore, a small number of leaf nodes will facilitate interval query processing using the auxiliary 3D R-tree. For intermediate nodes, more redundancy is permitted in order to maintain good performance for timestamp and short-interval queries.

Figure 3.3 illustrates the process of insertion for intermediate nodes. The R*-tree [BKS+90] *choose subtree* function determines the node, let A, where the new entry is inserted. In case that the node is full, a version split occurs and all the live entries are copied to a new node A', with their $t_{start}$ modified to the insertion time. The new entry is inserted into A'. If the total number of (live) entries in A' is above $b \cdot P_{svo}$, a strong version overflow occurs which triggers a key split. The general process is similar to MVB-trees except that we do not consider at all strong version underflows (thus, there is no parameter $P_{svu}$ in MVR-trees). The reason for this choice is two-fold: (i) underflows in MVR-trees happen much less frequently than overflows; (ii) we will handle underflows by entry re-insertion, which may trigger block overflows in several other nodes. In the sequel, we abbreviate weak version underflows as *underflows* without causing any ambiguity.

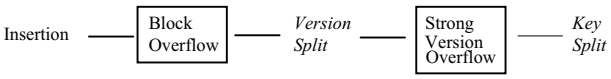



**Figure 3.3:** Insertion in intermediate nodes

Another difference from MVB-trees is that key and version splits need to take into account the spatial extents of the nodes. For key split we apply the R*-tree split function to minimize the overlap area of the two new nodes. For version splits we may need to tighten the MBRs of the new entries. This is illustrated in Figure 3.4. Assume that at timestamp 10, node A is version split, generating node B. Let $B_1$ be the entry duplicated from $A_1$; then $A_1$, $B_1$ point to the same node C. Since only two entries ($C_5$ and $C_6$) in node C are alive at timestamp 10 and bounded by $B_1$, the MBR of $B_1$ is probably much smaller than that of $A_1$. Meanwhile, since the lifespan of $A_1$ does not include timestamp 10, its MBR does not cover $C_6$, and may be tightened.

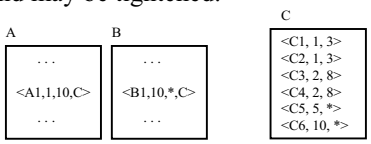



**Figure 3.4:** Tightening of MBRs

Insertion in leaf nodes, shown in Figure 3.5, is more complicated. The node is again selected by *choose subtree*, but block overflows are handled differently. In order to avoid version splits, we try the following alternatives (in this order): (i) *general key split*, (ii) insertion after reinserting one of the existing entries of the node, and (iii) insertion of the object in another node. Only if these alternatives fail, a version split occurs. In the sequel we describe each of the three techniques in detail.

G*eneral key split* is motivated by the fact that, in some cases a node to be version split can instead be key split without violating the weak version condition (recall that a key split does not generate redundancy). Consider, for example, the full node of Figure 3.6 ($b$=10, $P_{version}$=1/3). If a new entry is to be inserted, the entries can be distributed to two nodes so that for each timestamp in the range [1,*) there exist at least $b \cdot P_{version}$ entries alive. Thus, version split, which will generate version redundancy for entries $S_3$ to $S_{10}$, can be avoided. In many cases, however, it may be difficult, or impossible, to achieve such a partitioning of the entries. Furthermore, the two new nodes should have small overlap. The difference between general and ordinary key split is that the second one is applied when all the entries in the node to be split are alive and their $t_{start}$ equals the current time (which is the case after strong version overflows). Thus, the weak version condition can always be satisfied. Figure 3.7 shows the algorithm for general key split.

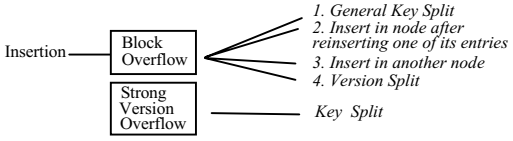



**Figure 3.5:** Insertion in leaf nodes

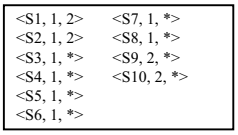



**Figure 3.6:** A node that can be general key split

Algorithm General_Key_Split(*this_node*)
1. *Choose split axis* $A_{split}$
2. sort all the entries according to their starting points on $A_{split}$
3. for ($k$ = 0 to $b$ + 1 – 2$b \cdot P_{version}$)
4. Distribute the first $k + b \cdot P_{version}$ entries into the first group and the rest into the second group
5. if both groups satisfy the version condition
6. *Overlap* = overlap area of the two groups
7. if *Overlap* < *min_Overlap*
8. *min_Overlap*=*Overlap*; record the distribution
9. sort all the entries according to their ending points on $A_{split}$ and goto line 3
10. return distribution with *min_Overlap* provided that *min_Overlap*<0.5·area of the original node's MBR

**Figure 3.7:** The general key split algorithm

If general key split fails, we try to reinsert an existing entry of the node, in order to make room for the new entry. This breaks the convention in multi-version structures (such as MVB-, BTR-, PPR-trees, etc.) that dead nodes cannot be modified. In MVR-trees, any leaf node can store a re-inserted entry provided that it satisfies the following conditions: (i) its lifespan must cover that of the entry; (ii) it should be dead if the entry is dead (we want to preserve live nodes for future insertions); (iii) its area should not be enlarged much, in order to ensure good performance for timestamp queries. In practice, only dead nodes with lifespans in the near past are modified.

The algorithm *choose subtree to reinsert*, shown in Figure 3.8, tries to find such a node that satisfies the

```
Algorithm Choose_Subtree_to_Reinsert
(this_node, entry_to_reinsert)
1.  if this_node.level=0 (i.e., a leaf)
2.      if this_node is not full return this_node
        /* entry_to_reinsert will be inserted into this node */
3.      else return failure
4.  list={all entries in this_node}
5.  if this_node level=1 and entry_to_reinsert is dead
6.      list=list-{live entries}
7.  list=list-{entries whose lifespans do not cover that of
    entry_to_reinsert}
8.  list=list-{entries with area enlargements greater than the
    threshold}
9.  sort list in ascending order of the area enlargement incurred
    by the reinsertion
10. list=list-{entries with area enlargement above the specified
    percentage w.r.t. the best entry}
11. for each entry e in list  Choose_Subtree_to_Reinsert (e,
    entry_to_reinsert); return success if a leaf node is found
12. return failure after all the entries in list have been searched
```

**Figure 3.8:** Choose subtree to reinsert

```
Algorithm Reinsert
1.  Retrieve all entries that can be reinserted without violating
    the version condition (note: taking the new entry into
    account) into set S_del
2.  Sort entries in S_del according to the benefits generated by
    their reinsertion
3.  while (S_del is not empty)
4.    entry_to_reinsert = the first entry in S_del
5.    remove entry_to_reinsert from S_del
6.    Choose_Subtree_to_Reinsert(root,
      entry_to_reinsert)
7.    if successful re-insertion then return success
8.    return failure
```

**Figure 3.9:** Reinsert algorithm

above conditions. The thresholds for area enlargement (line 8) are set to 1%, 0.1%, and 0% for nodes of level 0, 1 and higher. For instance, an entry can be re-inserted in a leaf node only if the enlargement is less than 1% of the original node area. Above level 1 the re-insertion should not cause any area enlargement. Furthermore (line 10), the area enlargement should not be worse than that of the best entry by a percentage (we use the same thresholds).

The *re-insertion* algorithm is presented in Figure 3.9. We first retrieve the set of entries that can be reinserted, i.e., the entries whose removal from the node will not lead to violation of the weak version condition. Then we sort these entries according to the benefits generated by their reinsertion. Dead entries always come before live ones. This is because live entries will be reinserted only into live nodes, which may also overflow and induce the same problem in the future. Among the dead and live entries, sorting is based on the area decrease of the node MBR caused by the entry deletion; entries that lead to larger area decrease are preferred. Finally each entry in the sorted list is retrieved, and *choose subtree to reinsert* is applied to check if it can be reinserted.

Notice that, unlike R*-trees, we reinsert a single entry, even if it is possible to reinsert more. This is due to the following reasons: (i) Reinsertion saves space but does not achieve structure improvement, as happens for R*-trees; (ii) Reinsertion of a single entry already achieves the objective of averting the version split. Any empty room in earlier nodes may be utilized by subsequent re-insertions to avoid future version splits.

The third heuristic tries to insert the new entry into another node that is not full. Specifically, we backtrack to the upper level and try to insert the entry into another branch. As before, in order to avoid query performance deterioration, we only consider branches that will incur small area enlargements: the area enlargements of candidate branches can only exceed that of the best branch by a certain percentage. In our implementation, we apply this heuristic for the two lowest levels of the tree with thresholds 1% and 0.1% (for levels 0 and 1 respectively). Using these values, on average 5% of the branches per node are attempted for most typical datasets.

We determined the order of the previous optimization heuristics by taking into account both the update cost and the quality of handling. General key split, which does not require reading any more pages, is the most efficient method in terms of update cost. Furthermore, it reduces the number of entries in the new nodes so that they will not overflow again in the near future; thus, it is the first method considered. The other two heuristics are more expensive. In particular, re-insertion of an existing entry can search multiple branches, while insertion in another (sub-optimal) node may require backtracking up to level 2. However, as we show in the experimental evaluation, their costs are compensated by the space savings. Only if the three previous heuristics fail, a version split occurs as the final alternative after a block overflow.

### 3.2 Deletion and underflow handling in MVR-trees

If a deletion does not incur structural changes, it is handled in a way similar to R*-trees. After the entry to be deleted is found, its $t_{end}$ is modified from * to the current time. An entry is physically deleted, only if its $t_{start}$ is equal to the current time (multiple updates may happen at the same timestamp). As with insertion, we follow two different policies for leaf and intermediate nodes when handling underflows caused by deletion.

Assume that an underflow occurs at the current timestamp $t$. For an intermediate node, we set the $t_{end}$ of its live entries to $t$. Then, these entries are re-inserted into the most recent logical R-tree after setting $t_{start}=t$. Notice that MVB-trees (also BTR- and PPR-trees) handle underflows by merging with sibling nodes, while MVR-trees apply the R*-tree algorithms.

For leaf nodes, in order to avoid redundancy caused by entry reinsertion, we first attempt to borrow a live entry from a sibling node. Consider, for instance, the two nodes

A, B and their parent S in Figure 3.10 and suppose that at timestamp 2, entry $A_1$ is deleted causing an underflow of node A. Instead of copying and reinserting the live entries of A immediately, we try to acquire a live entry from node B to "fill" A. The entry to be moved should have the following properties. First, it must be alive and its lifespan must be covered by that of A. In this example, all the live entries in B satisfy this condition. Second, after the removal of the entry, the version condition must still be satisfied in B. Notice that entries $B_2$ and $B_3$ cannot be moved because their deletion from B will cause weak version underflow for timestamp 1. Third, inserting this entry to node A will not cause its MBR to increase above a threshold. In our implementation we set this threshold to 1% of the original node area. Only if this heuristic fails, reinsertion is performed.

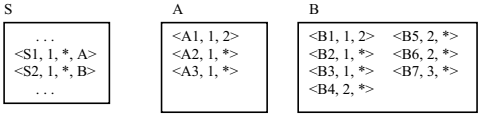



**Figure 3.10**: Borrowing a live entry from a sibling

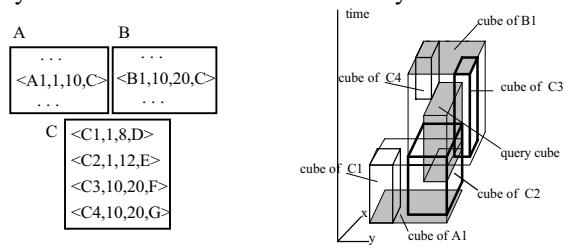
### 3.3 Construction of the auxiliary 3D R-tree

The auxiliary 3D R-tree is built on the leaves of the MVR-tree in order to process interval queries. Because, given a moderate node capacity, the number of leaf nodes in an MVR-tree is much lower than the actual number of objects, this tree is expected to be fairly small compared to a complete 3D R-tree. Its construction is straightforward: whenever a leaf node of the MVR-tree is updated, the change is propagated to its entry in the 3D R-tree. Adding the auxiliary tree not only improves interval query performance, but may also provide flexibility in other scenarios. An obvious example concerns spatio-temporal joins, an extension of spatial joins that retrieve all pairs of tuples satisfying some spatial condition during a time interval (or timestamp). Such queries can be processed by using the auxiliary trees and efficient R-tree join algorithms (e.g. [BKS93]).

### 3.4 Query processing with MV3R-trees

The combined structure gives two choices for query processing: the auxiliary 3D R-tree or the MVR-tree. For timestamp queries the MVR-tree is expected to perform better, while for long intervals the 3D R-tree is preferable. For short interval queries, selecting the appropriate tree calls for optimization methods that require analysis beyond the scope of this paper. We adopt the simple, but effective, heuristic that the 3D R-tree will be used whenever the temporal query length exceeds a certain threshold. Notice that the threshold may increase as time evolves, because a 3D R-tree's performance gradually deteriorates as the tree grows (while this is not a problem for the MVR-tree).

Querying with the auxiliary 3D R-tree is straightforward except that we treat MVR-tree nodes as the leaf level. Timestamp query processing using MVR-trees involves retrieval of the root whose jurisdiction interval covers the queried timestamp, and then search is performed similarly to R-trees. A problem arises for interval queries that may need to search multiple trees: we should avoid duplicate visits to the same node via different parents. Duplicate pointers to a node are created in version splits or entry re-insertions. In both cases the two entries pointing to the same node have disjoint lifespans. Consider the example of Figure 3.11a, where entries $A_1$ and $B_1$ point to the same child node C (due to a version split at timestamp 10). $A_1$ and $B_1$ are temporally adjacent, while $A_1$ spatially covers the entries (i.e., $C_1$, $C_2$) alive before, and $B_1$ the entries (i.e., $C_2$, $C_3$, $C_4$) alive after timestamp 10. Figure 3.11b shows the bounding boxes for the entries in 3.11a, as well as a query box. $C_2$ and $C_3$ (in bold lines) intersect the query box so their subtrees (nodes E and F) should be searched. Since node C may be reached twice (by following $A_1$ and $B_1$), we may attempt redundant visits to E and F. Such duplicate visits will result in severe IO cost as shown in [BS96], which solved the problem for MVB-trees using *reference points* and *backward pointers* between nodes. Unfortunately, these techniques are not applicable to MVR-trees. In BTR-trees and PPR-trees, a list is maintained in memory to keep the IDs of the visited pages. This approach, however, is not practical as it confines the maximum page accesses in a query to the amount of available memory.




(a) Duplicate pointers (b) Inferring future visits

**Figure 3.11:** Avoiding duplicate visits

Figure 3.12 describes an algorithm that avoids duplicate visits to the subtree of a node, but not to the node itself (e.g., C will be visited twice but E and F only once). The idea of the algorithm is to postpone visiting the branches of a shared node, if some branch will be visited in the future. Suppose, for instance that we have come to node C via $A_1$ (*this_node*=C, *parent_entry*=$A_1$). The entries of C will be checked (lines 1-2) in order to find one that intersects both $A_1$ and the query box. If no such entry exists, the algorithm backtracks to the previous level. Even if there are entries in C that intersect the query (but not $A_1$), they will be visited later. In our example, $C_2$ intersects $A_1$ and the query box, so the algorithm proceeds to line 4, at which point it has to decide whether to visit the qualifying subtrees now, or postpone it. Lines 4-6

check if there is any other entry *e* in the node that intersects the query and outlasts $A_1$ (which implies that it has a different parent). $C_3$ satisfies both conditions, meaning that C will be visited again in the future through $C_3$'s parent. Thus, the visits to E and F are postponed.

```
Algorithm Interval_query (this_node, parent_entry, query_box)
1.  for every entry e in this_ node
3.      if (e.box intersects parent_entry.box and query_box)
            goto line 4
3.  return /* no solutions at this level */
4.  for every entry e in this_ node
5.      if (e.box intersects query_box) and (e_end > parent_entry.t_end)
6.          return /* postpone visiting subtrees for later */
7.  for each entry e that intersects query_box
8.      Interval_query(e.pointer, e.box, query_box)
```

**Figure 3.12:** The interval query algorithm for MVR-trees

# 4. Experiments

In this section, we compare MV3R-trees with HR- and 3D R-trees through extensive experimentation. Due to the lack of real data, synthetic datasets with real-world semantics were generated by the GSTD method [TSN99]. GSTD has been employed (e.g., [NST99, PJT00, TP01]) as a benchmarking environment for access methods aimed at moving objects. For all the following experiments, the datasets contain regions with density 0.2. Unless otherwise stated, both the initial locations and movements of the objects are uniform. The spatial universe is a unit square, whereas timestamps are represented as integers. 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 average *p%* of the objects change their positions at each timestamp.

In order to simulate real life situations, we execute *workloads* with 500 timestamp and interval queries where the percentage of interval queries can be 0%, 20%, ..., or 100%. Notice that 0% implies a workload with only timestamp queries, and, similarly, 100% means only interval queries. In each workload, the order in which queries are applied is random. Cost is measured in terms of node accesses. Three parameters are taken into account when measuring the relative and absolute performance of the access methods, namely, the agility of the dataset, the spatial extents and the temporal lengths of the queries. For each of these factors we consider three representative values: (i) agilities 3%, 10%, and 20% represent *slow*, *medium*, and *fast* datasets; (ii) extents 0.5%, 2%, and 8% of the spatial universe, represent *small*, *medium* and *large* queries; (iii) query lengths 3.75%, 7.5%, 15% of the total number of timestamps represent *short*, *medium* and *long* queries. In order to generate these values, we set *max_length* to 7.5%, 15%, 30% respectively, and the query lengths distribute uniformly from 0 to *max_length.* All timestamps are queried with the same probability.

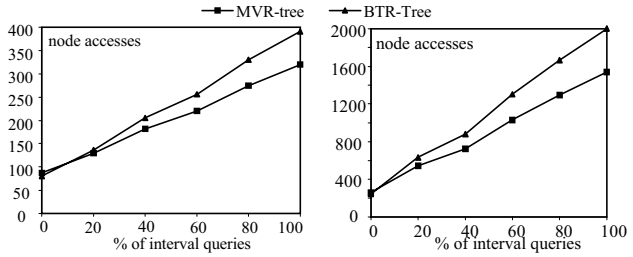
The R-tree implementations are based on R*-trees [BKS+90]. For HR-trees, overflows are always treated with splits, as entry reinsertion may cause considerable duplication. The page size is fixed to 1024 bytes for all cases. Using this size, the fanouts of HR-trees, 3D R-trees, and MVR-trees are 42, 36, and 36 respectively. The parameters of the MV3R-tree are as follows: $P_{version}$=0.35 and $P_{svo}$=0.85. Notice that $P_{svo}$ must be at least twice as large as $P_{version}$ (to ensure both the new nodes after a key split can satisfy the weak version condition). The last parameter specifies the maximum temporal length for a query to be answered by the MVR-tree. In this work, this parameter is 80% of the minimum jurisdiction interval of all the roots in an MV3R-tree. As a rough estimate this corresponds to 5% of the total number of timestamps. Longer intervals are answered by the auxiliary 3D R-tree.

### 4.1 Evaluation of heuristics for MV3R-trees

In the first set of experiments we demonstrate the effectiveness of the optimization heuristics employed for node overflows and underflows in MVR-trees. We use a medium agility ($p$=10) dataset with 50K objects at each timestamp, evolving for 200 timestamps. As a benchmark we implemented BTR-trees (designed for indexing bi-temporal data which is 2-dimensional; therefore the BTR--tree could be used for indexing regions). The size of the MVR-tree is 87.1 megabytes, while the BTR-tree is 124 megabytes. In order to compare the query performance, we substituted the MVR- with the BTR-tree in the MV3R-tree structure, and executed workloads with (i) queries with medium size and length, and (ii) large, long queries. Figure 4.1 shows the number of node accesses as a function of the percentile of interval queries using the above workload types. Obviously the optimization methods lead to significant improvements, which increase with the percentage of interval queries. The size reduction of MVR-trees (usually around 60%) with respect to BTR-trees results in a small number of leaf nodes to be indexed by the auxiliary 3D R-tree facilitating its efficiency on interval queries. As a side effect timestamp queries are slightly compromised, but the differences are very small.

Next, we also create slow and fast datasets and measure the percentage of cases that each heuristic was applied during updates. Table 4.1 shows the results for insertions, and Table 4.2 for deletions. For block overflows, *general key split* is applied infrequently. This is not surprising considering the tight conditions that a node must satisfy before it can be key split. However, since general key split is cheap and provides significant advantages when it occurs, we try to apply it whenever possible. *Reinserting an existing entry* handles the majority of the block overflows. Because reinsertion fills up the empty positions in the dead nodes, the tree tends to

be more compact. Notice that the frequency of reinsertion increases with the agility, which decreases the percentage of *insertions in another node* and *version splits*. In all cases the probability that a version split will happen after an overflow is less than 20%. According to Table 4.2, about ¼ of weak version underflows are treated by *borrowing another entry*, and the remaining by *reinsertion*, independently of the agility.




(a) First workload (b) Second workload

**Figure 4.1**: Comparison of MV3R-trees and BTR-trees

| | General key split | Reinsert an existing entry | Insert in another node | Version split |
|---|---|---|---|---|
| Low agility | 2.6 | 40 | 38.3 | 19.1 |
| Medium agility | 2.2 | 50.2 | 32.6 | 13 |
| High agility | 2 | 61 | 26.8 | 10.2 |

**Table 4.1**: Frequency (%) of heuristics after overflows

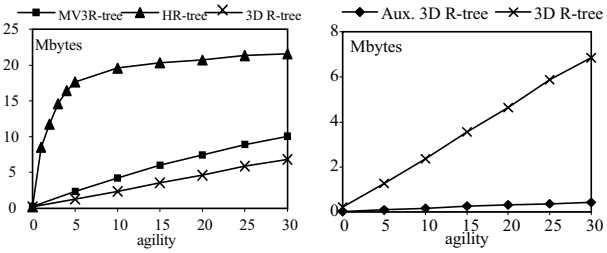
| | Borrow entry from neighbor | Reinsert |
|---|---|---|
| Low agility | 26 | 74 |
| Medium agility | 27 | 73 |
| High agility | 26.5 | 73.5 |

**Table 4.2:** Frequency (%) of heuristics after underflows

### 4.2 Comparison with HR- and 3DR-trees

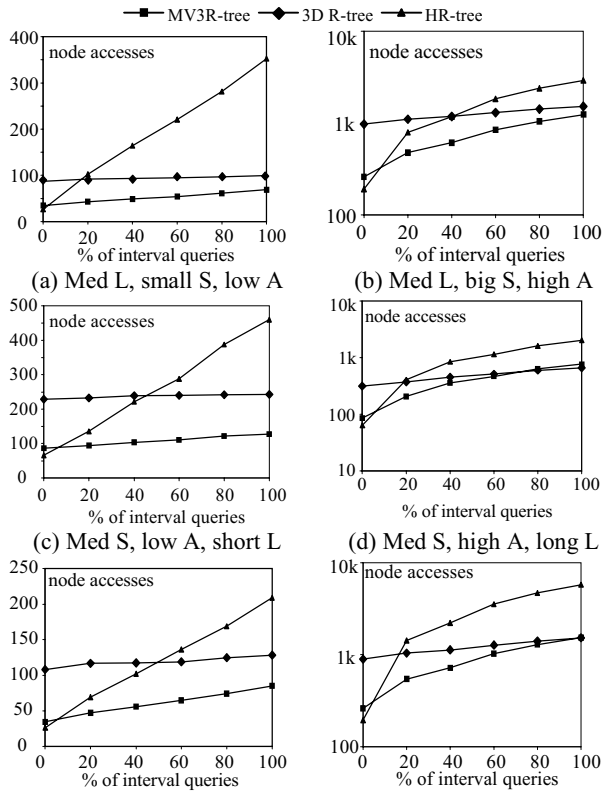
In order to compare the sizes of HR-trees, MV3R-trees, and 3D R-trees we created datasets with cardinality 5K and agilities in the range [1%, 30%]. Figure 4.2a shows the sizes of each structure after 100 timestamps as a function of agility. As expected, 3D R-trees have the smallest size, whereas HR-trees are the largest. MV3R-trees are about 1.5 times larger than 3D R-trees, and significantly smaller than HR-trees implying that much less redundancy is necessary in MV3R-trees. For agility values below 5, this difference is about an order of magnitude. Recall that, as discussed in section 2, each update in HR-trees can spawn many new nodes, which explains the inefficiency of the structure. For agility values above 10%, the sizes of HR-trees almost stabilize because the structure essentially degenerates to individual R-trees, one for each timestamp. Figure 4.2b compares the sizes of 3D R-trees with the auxiliary 3D R-trees built in MV3R-trees. The auxiliary tree is around 15 times smaller and accounts for only 3% of the size of the corresponding MV3R-tree.

In order to evaluate the query performance for a wide range of situations, we compared the access methods under various settings of agility, window sizes and interval lengths. All the following experiments are executed over datasets of 50K objects that evolve for 200 timestamps. Figure 4.3 shows the cost of each structure as a function of the interval query percentage in the workload. In each diagram, we set one of the above parameters to its medium value, and test the two extreme values for the other parameters. In Figures 4.3a and 4.3b, for instance, we use workloads with medium query lengths and test respectively (i) small query windows and slow datasets, and (ii) large query windows and fast datasets.




(a) All methods (b) Aux. vs normal 3D R-trees

**Figure 4.2**: Size comparison




(a) Med L, small S, low A (b) Med L, big S, high A

(c) Med S, low A, short L (d) Med S, high A, long L

(e) Med A, small S, short L (f) Med A, big S, long L

A=agility, S=size, L=length

**Figure 4.3**: Query performance comparison

As shown in the Figure 4.3, MV3R-Trees have the best overall performance. HR-trees are slightly better for timestamp queries, but the difference does not justify the excessive space requirements and poor interval query performance of this structure. MV3R-Trees consistently

outperform 3D R-trees even when all queries are intervals. The only case that 3D R-trees are better occurs at the right end of Figure 4.3d, where both the agility and the query length have their maximum values. This is because, high agility increases the redundancy (and the size) of the MVR-tree while the longer the query's interval is, the more redundancy will be retrieved. In summary, MV3R-trees compare favourably with HR-trees and 3D R-trees even in the extreme cases of only timestamp or only interval queries. The real strength of the new structure, however, does not lie in the extreme cases, but in the rather prevalent situation where a spatiotemporal system has to handle both types of queries.

## 5. Conclusions and future work

This paper addressed the problem of the indexing and retrieval of moving regions' past locations by proposing the MV3R-tree, a structure that combines the concepts of MVB-trees and 3D R-trees. Extensive experiments have shown that MV3R-trees can handle both timestamp and interval queries efficiently with relatively small space requirements. The current implementation of MV3R-trees could be further improved on the following aspects: (i) analytical cost models for determining the optimal (MVR- or 3D R-) tree to answer short interval queries, and (ii) (iii) overflow and underflow handling heuristics that are more efficient in terms of update cost, and can avert more version splits.

Although spatio-temporal databases have received extensive attention in the past few years, many problems remain unsolved. As mentioned earlier, various applications place different demands on the indexing structures. Existing access methods, for example, are not efficient for scenarios where most of the objects are moving 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 the time just handling the updates. Furthermore, this would result in huge space requirements. A better approach is to store information about objects' motion patterns (e.g. velocities). Such a solution would require novel indexing structures and query processing techniques.

## Acknowledgements

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

## References

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

[BKK+90] Burton, F., Kollias, J., Kollias, V., Matsakis, D. Implementation of Overlapping B-trees for Time and Space Efficient Representation of Collection of Similar Files. *The Computer Journal* 33(3): 279-280, 1990.

[BKS+90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD*, 1990.

[BKS93] Brinkhoff, T., Kriegel, H.P., Seeger, B. Efficient Processing of Spatial Joins Using R-trees. *ACM SIGMOD*, 1993.

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

[FGN+00] Forlizzi, L., Güting, R., Nardelli, E., Schneider, M. A Data Model and Data Structures for Moving Objects Databases. *ACM SIGMOD,* 2000.

[KGT99] Kollios, G., Gunopulos, D., Tsotras, V. On Indexing Mobile Objects. *ACM PODS,* 1999.

[KGT+01] Kollios, G., Gunopulos, D., Tsotras, V., Delis, A., Hadjieleftheriou, M. Indexing Animated Objects Using Spatiotemporal Access Methods. To appear in *IEEE TKDE*.

[KS91] Kolovson, C., Stonebraker, M. Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data. *ACM SIGMOD*, 1991.

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

[NS98] Nascimento, M., Silva, J. Towards Historical R-trees. *ACM SAC*, 1998.

[NST99] Nascimento, M., Silva, J., Theodoridis, Y. Evaluation of Access Structures for Discretely Moving Points. *International Workshop on Spatio-Temporal Database Management*, 1999.

[PJT00] Pfoser, D., Jensen, C., Theodoridis, Y. Novel Approaches to the Indexing of Moving Object Trajectories. *VLDB*, 2000.

[SJL+00] Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M. Indexing the Positions of Continuously Moving Objects. *ACM SIGMOD*, 2000.

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

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

[TSP+98] Theodoridis, Y., Sellis, T., Papadopoulos, A., Manolopoulos, Y. Specifications for Efficient Indexing in Spatiotemporal Databases. *IEEE SSDBM*, 1998.

[TP01] Tao, Y., Papadias, D. Efficient Historical R-trees. *IEEE SSDBM*, 2001.

[VV97] Varman, P., Verma, R An Efficient Multiversion Access Structure. *IEEE TKDE* 9(3): 391-409, 1997.

[XHL90] Xu, X., Han, J., Lu, W. RT-tree: An Improved R-tree Index Structures for Spatiotemporal Data. *Int'l Symp. on Spatial Data Handling*, 1990.