An Adaptive Indexing Technique Using Spatio-Temporal Query Workloads

Hyung-Ju Cho, Jun-Ki Min, Chin-Wan Chung

Dept. of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology
373–1 Kusong-dong, Yusong-gu, Taejon 305–701, Korea
{*hjcho, jkmin, chungcw*}*@islab.kaist.ac.kr*

Abstract

Many spatio-temporal access methods, such as the HR-tree, the 3DR-tree, and the MV3R-tree, have been proposed for timestamp and interval queries. However, these access methods have the following problems: the poor performance of the 3DR-tree for timestamp queries, the huge size and the poor performance of the HR-tree for interval queries, and the large size and the high update cost of the MV3R-tree. We address these problems by proposing an adaptive partitioning technique called the Adaptive Partitioned R-tree (APR-tree) using workloads with timestamp and interval queries. The APR-tree adaptively partitions the time domain using query workloads. Since the time domain of the APR-tree is automatically fitted to query workloads, the APR-tree outperforms the other access methods for various query workloads. The size of the APR-tree is on the average 1.3 times larger than that of the 3DR-tree which has the smallest size. The update cost of the APR-tree is on the average similar to that of the 3DR-tree which has the smallest update cost.

Key words: Spatio-Temporal Databases, Indexing Technique, R-trees, Timestamp and Interval Queries

1 Introdcution

Spatio-temporal databases manipulate moving objects whose positions and shapes may change along with time. In other words, they are said to be the combination of spatial databases and temporal databases. To support trajectory (or navigational) queries, timestamp (or timeslice) queries, and interval queries, many spatio-temporal access methods have been proposed. They can be classified into two groups. One consists of access methods for future queries,

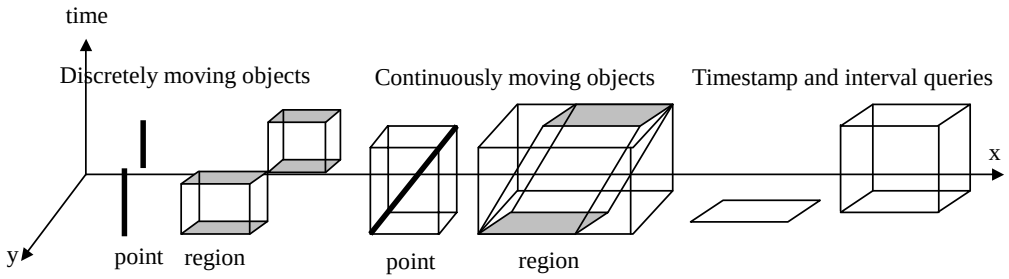


Fig. 1. Moving objects and timestamp/interval queries

and the other for historical queries. Although timestamp and interval queries are available to retrieve the future information of moving objects, in this paper, we focus on timestamp and interval queries for the past information. Note that time is represented as discrete timestamps [3,8,9,11]. Timestamp queries retrieve all the objects intersecting a spatial window at a specific timestamp, and interval queries include multiple consecutive timestamps. Therefore, the timestamp query can be considered as a special case of the interval query, where the interval contains only one timestamp. Access methods for future queries focus on predicting the future locations of objects by storing their current positions, velocities, and directions [5–7]. Trajectory queries mainly deal with the navigational trajectories of particular objects. The STR-tree [8] and the TB-tree [8] belong to the access methods for trajectory queries.

Moving objects can be classified according to the spatial shapes and the temporal movements of objects [1,9,11,12]. Their spatial shapes are approximated by minimum bounding rectangles (MBRs). Their temporal movements are either discrete or continuous. However, the problem of indexing moving objects can be reduced to indexing discrete rectangles that contain both their spatial shapes and time intervals. Figure 1 illustrates this. The R-tree can be a good candidate for indexing these rectangles. Timestamp and interval queries can also be represented as rectangles. Therefore, timestamp and interval queries can be considered as range queries of the R-tree. However, the time region monotonically increases along with time while the spatial region does not. Therefore, if a single R-tree covers the entire time region, the performance of the R-tree would degrade for timestamp and interval queries since the time region enlarges as time passes.

In this paper, we concentrate on timestamp and interval queries for moving objects with historical information such as vehicles with GPSs, people with mobile devices, planes and ships on a voyage. Many spatio-temporal index structures have been developed for timestamp and interval queries during the past few years [3,10,11]. If a single index structure covers the entire time region, its performance would degrade since the indexed time region enlarges as time passes. Therefore, some spatio-temporal access methods partitions the whole time region into multiple time regions indexed by the corresponding R-

trees. However, since no existing spatio-temporal access methods utilize query workoads, they can neither support a variety of query workloads efficiently nor reflect the change of the query workload along with time. We address these problems by proposing the Adaptive Partitioned R-tree (APR-tree) which automatically adapts to workloads with timestamp and interval queries. Based on query workloads, the APR-tree adaptively partitions the time region. Our contributions are summarized as follows:

- The APR-tree achieves the best performance for various workloads with timestamp and interval queries except for workloads with 100% timestamp queries. Its performance is on the average 40% better than that of the MV3R-tree.
- Although the size of the APR-tree is affected by query workloads, the size of the APR-tree is on the average 1.3 times larger than that of the 3DR-tree which has the smallest size and does not exceed 2 times that of the 3DR-tree.
- Although the update cost of the APR-tree are also affected by query workloads, the update cost of the APR-tree is on the average similar to that of the 3DR-tree which has the smallest update cost.
- Through mathematical analyses and extensive experimental study, we show that the APR-tree deals with various query workloads more efficiently than the other access methods.

The rest of the paper is organized as follows: Section 2 surveys the existing access methods for timestamp and interval queries. Section 3 describes the APR-tree, and the corresponding insertion and query processing algorithms. Section 4 provides the optimal jurisdiction interval length of the APR-tree based on mathematical analyses of the performance and the size of the APR-tree. Section 5 presents experimental results over various query workloads. Finally, Section 6 provides conclusions.

2 Related Work

In this section, we describe structures, and strong and weak points of existing access methods, the 3DR-tree, the HR-tree, and the MV3R-tree.

2.1 3DR-trees

The 3DR-tree [10] simply considers time as another dimension of the R-tree [15,16]. Whenever an object moves to another position or changes its shape, a new MBR is created to represent the change of the object and the

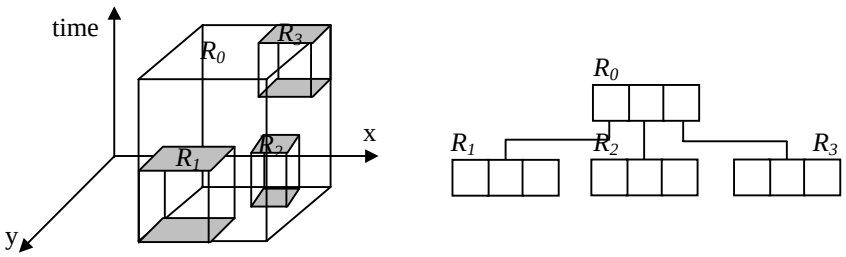


Fig. 2. An example of the 3DR-tree

MBR containing both its spatial extent and lifespan is inserted into the 3DR-tree. This MBR is effective up to the next change of the object. Figure 2 shows an example of the 3DR-tree. In this figure, R_1, R_2, and R_3 represent MBRs for the movements of objects, and R_0 contains R_1, R_2 and R_3. As time passes, the time region enlarges accordingly. This makes the performance of the 3DR-tree degrade because in the 3DR-tree, an R-tree keeps the whole time region.

Also, long-lived records cause a huge dead space which is a part of the MBR, but does not cover any record. Therefore, they degrade the query performance of the 3DR-tree. However, the 3DR-tree has the smallest size since the 3DR-tree has no duplicate data unlike the HR-tree and the MV3R-tree.

2.2 HR-trees

The Historical R-tree (HR-tree) [3] creates an R-tree whenever objects in the previous R-tree change their positions or shapes, but common branches of consecutive R-trees are stored only once in order to save space.

The timestamp query is directed to the corresponding R-tree and the search is performed inside this tree only. Thus the timestamp query becomes the ordinary window query and is handled very efficiently. The interval query should search the corresponding R-trees of all the timestamps involved. Even when only one object has changed its position, the HR-tree may update nodes contained in the path between the leaf node corresponding to the object and the root node. Therefore, the size of the HR-tree is several times larger than that of the 3DR-tree. Figure 3 shows an example of the HR-tree. $R_1[t_1, t_2)$ deals with records whose time intervals are contained in time interval $[t_1, t_2)$ and $R_2[t_2, t_3)$ manipulates records whose time intervals are contained in $[t_2, t_3)$. Suppose that object f changes its position to f' from B_0 to C_0 at t_2. The three nodes B_0, C_0, D_0 which are associated with f and f' are copied and updated to three new nodes B_1, C_1, D_1 in the gray color. Note that the unchanged other objects except f and f' are shared between R_1 and R_2.

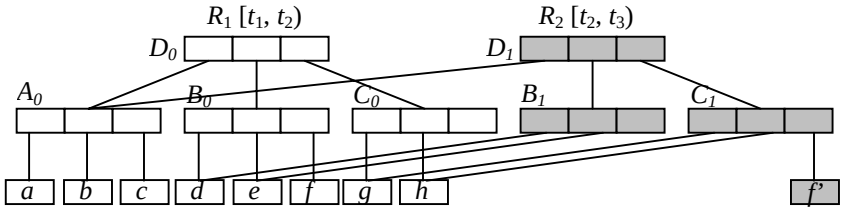


Fig. 3. An example of the HR-tree

2.3 MV3R-trees

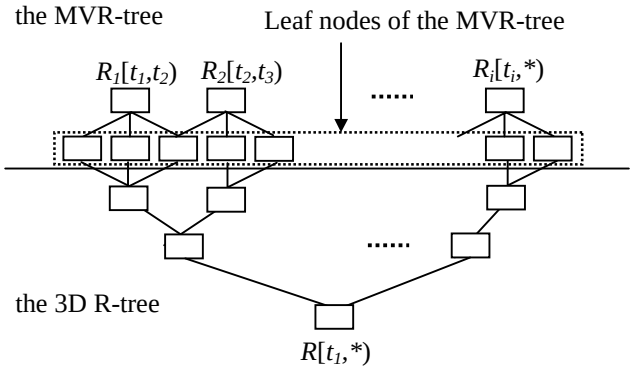


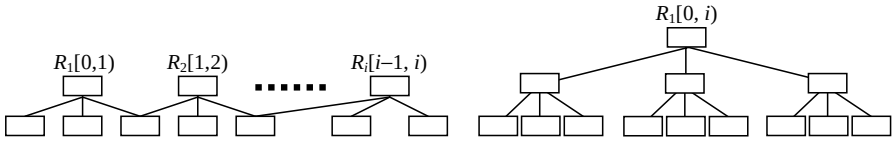
Fig. 4. The structure of the MV3R-tree

The Multi-Version 3DR-tree (MV3R-tree) [11] is the access structure that combines the Multi-Version R-tree (MVR-tree) and a small auxiliary 3DR-tree built on the leaf nodes of the MVR-tree. The former is used to answer timestamp and short interval queries and the latter to answer long interval queries. Figure 4 illustrates the structure of the MV3R-tree. As shown in this figure, the MVR-tree consists of multiple R-trees that have their own jurisdiction intervals. For example, $R_1[t_1, t_2)$ deals with records whose time intervals belong to $[t_1, t_2)$ while $R_2[t_2, t_3)$ manipulates records whose time intervals belong to $[t_2, t_3)$.

Although the size of the auxiliary 3DR-tree is very small since it shares the leaf nodes of the MVR-tree, it improves the performance on interval queries and provides flexibility to algorithms for processing other spatial queries such as the join and the k-nearest neighbor.

However, the MV3R-tree has a few disadvantages. The size of the MV3R-tree is typically 1.5 times larger than that of the 3DR-tree due to many duplicate data from version splits [4]. In addition, the update cost of the MV3R-tree is much higher than those of the HR-tree and the 3DR-tree. They maintain one structure, while the MV3R-tree has to modify two structures simultaneously when records are inserted. The auxiliary 3DR-tree suffers from the large search space for interval queries since it is also responsible for the whole history like the 3DR-tree.

One of the most significant differences between the HR-tree and the 3DR-tree is the average jurisdiction interval length of R-trees which form each structure. In the HR-tree, R-trees are almost created every timestamp and each R-tree is responsible for the information of the specific timestamp only. In the 3DR-tree, an R-tree is created only once, and this R-tree keeps all the information of the whole time range. Figure 5 illustrates this observation.



(a) an HR-tree (b) a 3DR-tree

Fig. 5. Structures of the HR-tree and the 3DR-tree

In the HR-tree, the timestamp query is evaluated by the R-tree which corresponds to the timestamp of this query. In the 3DR-tree, however, the timestamp query is always processed by a single R-tree that covers the whole time range. Thus, the 3DR-tree has to access much more nodes than the HR-tree due to a prohibitive search space. However, to answer interval queries, the HR-tree accesses more nodes than the 3DR-tree since the HR-tree has to search many R-trees whose jurisdiction intervals overlap these interval queries. The 3DR-tree should still search the largest space irrespective of the query condition. Consequently, to overcome the poor performance of the 3DR-tree on timestamp and interval queries, it is necessary to reduce the search space, while to overcome the poor performance of the HR-tree on interval queries, it is necessary to decrease the number of R-trees which are searched.

3 APR-Trees

Section 3.1 presents an adaptive partitioning method based on query workloads. Next, we describe *insert* and *search* algorithms of the APR-tree in Section 3.2.

3.1 Query Adaptive Partitioning Method

To overcome the problem with the 3DR-tree, we apply the query-adaptive partitioning method to the 3DR-tree. We call the resultant structure the Adaptive Partitioned R-tree (APR-tree). The APR-tree consists of multiple 3DR-trees,

each of which is responsible for the fixed time interval derived from query workloads. Since several new terms and symbols are used in this work, they are summarized in Table 1.

Table 1
Summary of symbols and terms

Symbol or Term	Description
*	the special reserved word "*now*".
$AVG_{records}$	the average lifespan length of records.
$AVG_{queries}$	the average time interval length of queries.
$R_i[t_i, t_{i+1})$	the R-tree that is created the i-th and covers the time interval of $[t_i, t_{i+1})$.
L_i	the jurisdiction interval length of $R_i[t_i, t_{i+1})$. Namely, $L_i = t_{i+1} - t_i$.
K	the number of R_i's.
C_i	the cardinality of objects with distinct identifiers which belong to R_i.
F	the average fanout of a node.
H_i	the height of R_i.
N_i	the number of records in R_i.
N	the total number of records.
M_h	the number of nodes at height h.
a_h	the average area covered by a node at height h.
D	the number of dimensions.
$Size(R-tree)$	the size of the R-tree.
$AJIL(R-tree)$	the average length of the time ranges covered by R-trees in the structure.

The average jurisdiction interval length (AJIL) is introduced to explain the property of the APR-tree. $AJIL(R-tree)$ can be defined as follows:

$$AJIL(R-tree) = \frac{1}{K} \cdot \sum_{i=1}^{K} L_i$$

where K is the number of R_i's and L_i is the jurisdiction interval length of R_i. In the case of the MVR-tree in Figure 6, $AJIL(MVR-tree) = \frac{10+11+10+9}{4} = 10$ since K is 4, and L_1, L_2, L_3, and L_4 are 10 (=10−0), 11 (=21−10), 10 (=31−21), and 9 (=40−31), respectively. Since $K = 1$ in the 3DR-tree, $AJIL(3DR-tree) = 40$ (=40−0). In the HR-tree, $AJIL(HR-tree) \cong 1$ since R-trees may be created every timestamp.

Table 2 summarizes $AJIL(R-tree)$ of each structure. The search space of queries is affected by $AJIL(R-tree)$ of each structure. In the case of timestamp queries, $AJIL(R-tree) = 1$ is optimal since the timestamp query is associated with the specific timestamp. In this reason, the HR-tree achieves the best performance for timestamp queries. Similarly, in the case of interval queries, $AJIL(R-tree)$ is closely associated with the interval query cost. The relationship between the interval query cost and $AJIL(R-tree)$ is explained in Section 4.3.

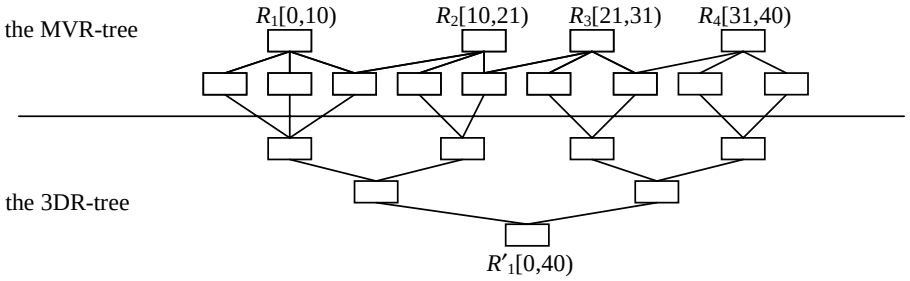


Fig. 6. $AJIL(MVR-tree) = 10$ and $AJIL(3DR-tree) = 40$

Table 2
$AJIL(R-tree)$ of each access method

Access Method	$AJIL(R-tree)$
HR-tree	1
3DR-tree	L_1
MV3R-tree	MVR-tree: $\frac{1}{K} \cdot \sum_{i=1}^{K} L_i$ Aux. 3DR-tree: L_1
APR-tree	$\frac{1}{K} \cdot \sum_{i=1}^{K} L_i$

Although $AJIL(APR-tree)$ is similar to $AJIL(MVR-tree)$, the APR-tree and the MVR-tree employ the very different policies of determining L_i. While L_i of the MVR-tree is determined by version splits of root nodes, L_i of the APR-tree is determined by query workloads. In other words, $AJIL(MVR-tree)$ depends on datasets, but $AJIL(APR-tree)$ depends on query workloads. Figure 7 depicts the structure of the APR-tree. Unlike the HR-tree and the MVR-tree, adjacent R-trees of the APR-tree do not share nodes. In this figure, let $A =< id, s, [t_{start}, t_{end}) >$ be a record for an object's movement, where id is an object identifier, s is the spatial MBR, and t_{start} and t_{end} are the time when the record was inserted and logically deleted, respectively. The logical deletion is to change the end time * (i.e. "*now*") of the lifespan of a live record to a specific time, where a live record denotes the record whose lifespan's end time is *. Since $A =< id, s, [t_{start}, t_{end}) >$ intersects the boundary between R_1 and R_2, it is divided into $A_1 =< id, s, [t_{start}, t_2) >$ and $A_2 =< id, s, [t_2, t_{end}) >$, and A_1 and A_2 are inserted into R_1 and R_2, respectively. Therefore, a duplicate record occurs when the lifespan of a record intersects the boundary.

In Figure 7, L_1, L_2, and L_i denote the jurisdiction interval lengths of R_1, R_2, and R_i, respectively, and the jurisdiction interval length of an R-tree may not be the same as those of other R-trees. This is due to the fact that the query workload may change along with time.

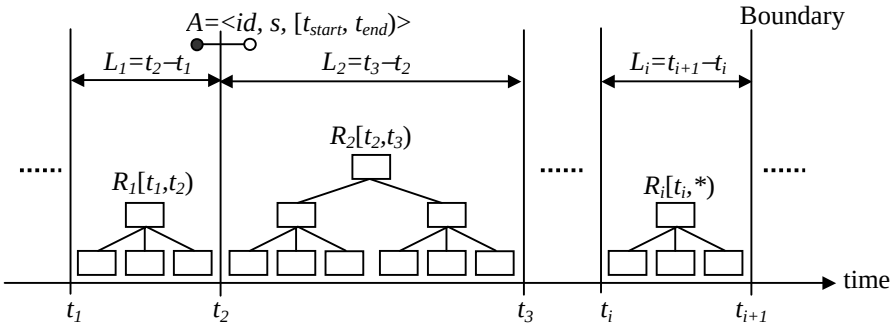


Fig. 7. The structure of the APR-tree

proc insert (record $r_2 =< id, s_2, [t_2, *) >$)
begin
1. /* find R_i which contains r_2 and insert r_2 into R_i */
2. **for** $R_i[t_i, t_{i+1})$ where i is from 1 to K **do**
3. **if** ($[t_i, t_{i+1})$ contains $[t_2, *)$) **then**
4. insert r_2 to R_i
5. **break** /* exit **for** loop */
6. **endif**
7.
8. /* conduct the logical deletion of r_1 that is the previous record of r_2 */
9. let $r_1 =< id, s_1, [t_1, *) >$ be the previous record of r_2
10. * of r_1 is changed into t_2. i.e., $r_1 =< id, s_1, [t_1, t_2) >$.
11.
12. /* find R_i's which intersect r_1 and insert r_1 into R_i's */
13. **for** $R_i[t_i, t_{i+1})$ where i is from 1 to K **do**
14. **if** ($[t_i, t_{i+1})$ intersects $[t_1, t_2)$) **then**
15. insert r_1 to R_i
16. **endif**
end

Fig. 8. Insertion algorithm of the APR-tree

3.2 Insertion and Search Algorithms

If the lifespan of a record does not intersect the boundary, the record is inserted into the R-tree covering the lifespan of the record. Otherwise, after the record is divided by the boundary (this is called the data fragmentation), each of records spawned by the data fragmentation is inserted into the R-tree whose jurisdiction interval contains the lifespan of the record. Figure 8 presents the insertion algorithm of the APR-tree. Note that the insertion of a record accompanies the logical deletion of the previous record.

Figure 9 illustrates an example of logical deletion. Record A_1 is logically

deleted at timestamp t_2, where s_1 and s_2 indicate the spatial MBRs of A_1 and A_2, respectively, and $[t_1, t_2)$ and $[t_2, *)$ indicate the lifespans of A_1 and A_2, respectively.

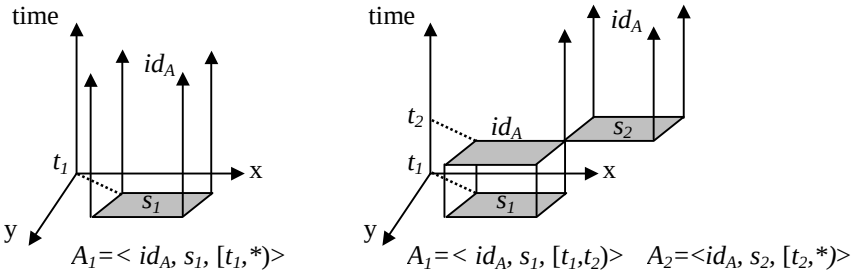


Fig. 9. Logical deletion at timestamp t_2

The timestamp and interval query processing with the APR-tree is similar to that with the 3DR-tree. However, since the 3DR-tree consists of a single tree structure, it should search the largest space. Particularly, this prohibitively degrades the performance of the 3DR-tree for timestamp queries. The APR-tree outperforms the 3DR-tree for timestamp and interval queries due to the search space restriction and the reduced dead space by the data fragmentation. Like the HR-tree and the MVR-tree, the APR-tree may search several R-trees whose jurisdiction intervals overlap the time interval of the query. The search algorithm of R-trees which constitute the APR-tree is the same as that of the original R-tree [15,16]. As shown in Figure 10, query Q_1 is processed by R_i and query Q_2 is processed by R_i and R_{i+1}.

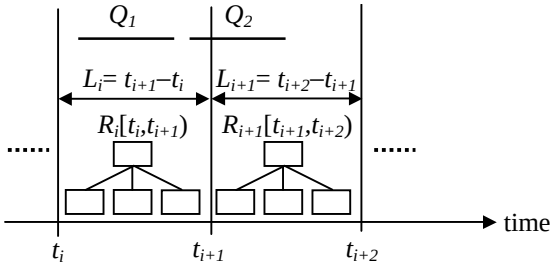


Fig. 10. Interval query processing in the APR-tree

4 Analysis

In this section, we assume that all the data are uniformly distributed on the d-dimensional work space $WS = [0, 1)^d$ which contains spatial and temporal dimensions ("uniformity assumption" [2,14,17]).

When the whole time domain is partitioned into the K time domains, the size of the APR-tree becomes the sum of the sizes of R-trees that constitute the APR-tree. Let $Size(R-tree)$ denote the size of the R-tree. The size of the APR-tree is:

$$Size(APR-tree) = \sum_{i=1}^{K} Size(R_i) \tag{1}$$

To compute $Size(R_i)$, we devise the formula $Size(R_i)$ as follows:

$$Size(R_i) = \sum_{h=1}^{H_i} \frac{T_i \cdot C_i}{f^h} \tag{2}$$

where T_i is the average number of records which keep the history of a moving object in R_i, C_i is the cardinality of moving objects in R_i, f denotes the average fanout of a node, and H_i denotes the height of R_i. Thus, $T_i \cdot C_i$ becomes the total number of records in R_i.

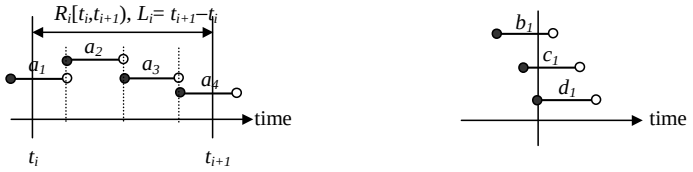
T_i can be computed by L_i and $AVG_{records}$, where L_i is the jurisdiction interval length of R_i and $AVG_{records}$ is the average lifespan length of records.

$$T_i = \frac{L_i}{AVG_{records}} + \left(1 - \frac{1}{AVG_{records}}\right) \tag{3}$$

In Formula (3), $\frac{L_i}{AVG_{records}}$ is the average number of non-duplicate records of a moving object in R_i and $\left(1 - \frac{1}{AVG_{records}}\right)$ is the average number of duplicate records of the moving object in R_i. Whenever an object changes its position or shape, a new record is inserted into the R-tree. Therefore, the average number of records spawned by the object is $\frac{L_i}{AVG_{records}}$. In Figure 11(a), let a_1, a_2, a_3 and a_4 be the four records spawned by an object. a_1 and a_4 should be divided due to the boundary, so two duplicate records occur. However, since the boundary is shared between adjacent R-trees, one of two duplicate records belongs to R_i and the other belongs to R_{i-1} or R_{i+1}. Additionally, as shown in Figure 11(b), since the timestamp is discrete, there are finite cases that a record crosses the boundary. When the start time of a record's lifespan meets the boundary such as record d1, the record is not fragmented. Therefore, an object produces $\left(1 - \frac{1}{AVG_{records}}\right)$ duplicate records.

Height H_i of R_i can be computed using a node's average fanout f and the total number of records $T_i \cdot C_i$ as follows [2]:

$$H_i = \left\lceil \log_f(T_i \cdot C_i) \right\rceil \tag{4}$$



(a) a_1 and a_4 are fragmented (b) d_1 is not fragmented

Fig. 11. Fragmentation of records by boundaries

From Formulas (1), (2), (3), and (4), the size of the APR-tree is:

$$Size(APR-tree) = \sum_{i=1}^{K} \sum_{h=1}^{H_i} \frac{\left(\frac{L_i}{AVG_{records}} + \left(1 - \frac{1}{AVG_{records}}\right)\right) \cdot C_i}{f^h} \qquad (5)$$

Let λ_i be the ratio of duplicate records in R_i. Then, λ_i can be defined as follows:

$$\lambda_i = \frac{N_{dup}}{N_{total}} = \frac{\left(1 - \frac{1}{AVG_{records}}\right) \cdot C_i}{\left(\frac{L_i}{AVG_{records}} + \left(1 - \frac{1}{AVG_{records}}\right)\right) \cdot C_i} \leq \frac{1}{\frac{L_i}{AVG_{records}} + 1} \qquad (6)$$

where N_{dup} is the number of duplicate records in R_i and N_{total} is the total number of records in R_i.

In Formula (6), L_i determines λ_i since $AVG_{records}$ is the average lifespan length of records and is obtained from records. Consequently, the larger L_i, the smaller λ_i. In terms of the size of the APR-tree, the larger L_i results in the smaller size of the APR-tree. Extremely, when $L_i = \infty$, the APR-tree has no duplicate records. Nevertheless, we should allow duplicate records for reducing the search space of timestamp and interval queries.

4.2 Estimation of the query cost of the APR-tree

To estimate the interval query cost of the APR-tree, we extend the range query cost model of the R-tree [2,15,18] since interval queries can be considered as range queries in the R-tree. When an interval query overlaps the jurisdiction interval of R_i, we estimate the number of nodes accessed to answer this interval query in R_i. Assume that $L_i = \frac{1}{K}$ since the number of R_i's is K.

Let M_h denote the number of nodes at height h and N_i denote the total number of records in R_i. M_h can be computed by N_i and as follows [2]:

$$M_h = \left\lceil \frac{N_i}{f^h} \right\rceil \text{ where } N_i = \left(\frac{L_i}{AVG_{records}} + \left(1 - \frac{1}{AVG_{records}}\right)\right) \cdot C_i \qquad (7)$$

For example, for the leaf level, $M_1 = \left\lceil \frac{N_i}{f} \right\rceil$.

Let a_h denote the average area covered by a node at height h. Since R_i covers 1/K of the d-dimensional work space $WS = [0,1)^d$ as shown in Figure 12, we have:

$$a_h = \frac{1}{K} \cdot \frac{1}{M_h} \tag{8}$$

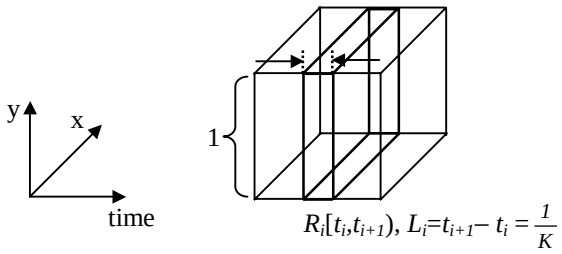


Fig. 12. R_i in $WS = [0,1)^d$

Using the Minkowski sum technique [2,14,17], the probability that a node at height h may overlap the interval query rectangle s is:

$$\left(\sqrt[d]{s} + \sqrt[d]{a_h}\right)^d \tag{9}$$

where $\sqrt[d]{s}$ and $\sqrt[d]{a_h}$ denote the average one-side lengths of s and a_h, respectively. Therefore, the number of nodes at height h that may overlap the query rectangle s is:

$$M_h \cdot \left(\sqrt[d]{s} + \sqrt[d]{a_h}\right)^d = \left(\sqrt[d]{\left\lceil \frac{N_i}{f^h} \right\rceil} \cdot s + \sqrt[d]{\frac{1}{K}}\right)^d \tag{10}$$

where M_h and a_h are substituted with N_i, f^h, and K by Formulas (7) and (8).

Let NA_i be the total number of nodes accessed to answer the query rectangle s in R_i. NA_i is computed by summing Formula (10) from the leaf to the root as follows:

$$NA_i = 1 + \sum_{h=1}^{\lceil \log_f N_i \rceil - 1} \left(\sqrt[d]{\left\lceil \frac{N_i}{f^h} \right\rceil} \cdot s + \sqrt[d]{\frac{1}{K}}\right)^d \tag{11}$$

When the interval query intersects the boundary, all the R_i's which overlap the interval query should be searched. Therefore, the query cost of the APR-tree is the sum of query costs of these R_i's which overlap the interval query.

In this section, we present how to determine the tuning parameter L_i which affects the query cost and the size. This procedure is based on the mathematical rationale using Formulas (6) and (11).

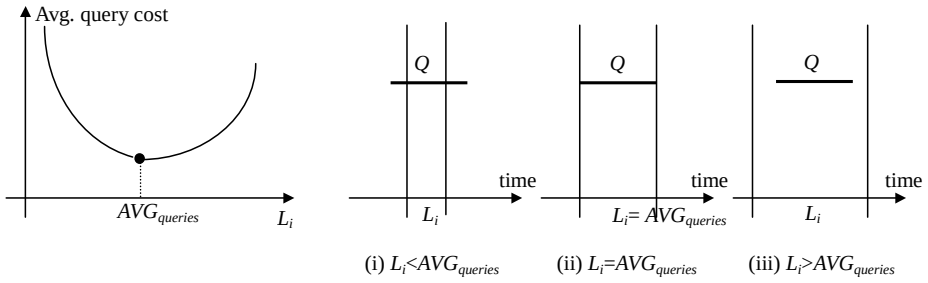


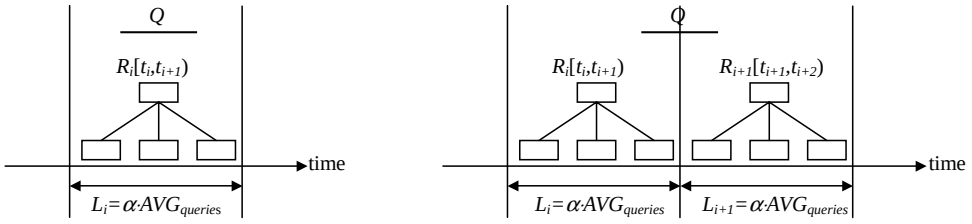
Fig. 13. Average query cost for interval query Q as a function of L_i

Figure 13 shows the average number of nodes accessed to answer an interval query Q whose time interval length is $AVG_{queries}$ as a function of L_i. If the interval query Q does not intersect the boundary, $L_i = AVG_{queries}$ makes the query cost smallest since the search space is smallest. When $L_i < AVG_{queries}$, the query cost increases since many R_i's should be searched. Conversely, when $L_i > AVG_{queries}$, the query cost also increases due to the increase of the search space. In other words, if $L_i < AVG_{queries}$, the query cost is affected by the number of R_i's which are searched. Since $L_i < AVG_{queries}$, the number of R_i's which are searched is at least 2. The smaller L_i decreases the search space, but increases the number of R_i's which are searched. In contrast, if $L_i > AVG_{queries}$, the query cost is affected by the search space. Since $L_i > AVG_{queries}$, the number of R_i's which are searched is at most 2. Consequently, if $L_i \leq AVG_{queries}$, L_i is chosen to minimize the number of R_i's which are searched. If $L_i \geq AVG_{queries}$, L_i is chosen to minimize the search space.

Let $L_i = \alpha \cdot AVG_{queries}$ ($\alpha > 0$). We can consider the following two cases according to the range of α: (i) $\alpha \leq 1$ (ii) $\alpha \geq 1$. Namely, $\alpha \leq 1$ means that $L_i \leq AVG_{queries}$, and $\alpha \geq 1$ means that $L_i \geq AVG_{queries}$.

If $\alpha \leq 1$ (i.e., $L_i \leq AVG_{queries}$), the query cost depends on the number of R_i's which are searched. Since the smaller L_i increases the number of R_i's which are searched, $\alpha = 1$ (i.e., $L_i = AVG_{queries}$) minimizes the number of R_i's which are searched. When $L_i = AVG_{queries}$, the number of R_i's which are searched is at most 2. Consequently, if $L_i \leq AVG_{queries}$, $L_i = AVG_{queries}$ minimizes the query cost.

If $\alpha \geq 1$ (i.e., $L_i \geq AVG_{queries}$), the query cost depends on the search space. As shown in Figure 14, there are the following two cases: The first case is that the interval query Q does not intersect the boundary. The second case is



(a) $S_{not\_intersect} = \alpha \cdot AVG_{queries}$ (b) $S_{intersect} = 2\alpha \cdot AVG_{queries}$

Fig. 14. Difference between search spaces of two cases

that the interval query Q intersects the boundary. Therefore, L_i is chosen to minimize the average search space of two cases.

Let $P_{intersect}$ be the probability that the interval query Q intersects the boundary. Since $P_{intersect}$ is the same as the probability that an interval whose length is $AVG_{queries}$ intersects a point within the range of $[0, L_i)$, we have:

$$P_{intersect} = \frac{AVG_{queries}}{L_i} \tag{12}$$

For example, if $L_i = 2 \cdot AVG_{queries}$, $P_{intersect} = 0.5$.

Let $P_{not\_intersect}$ be the probability that the interval query Q does not intersect the boundary. Since $P_{not\_intersect} = 1 - P_{intersect}$, we have:

$$P_{not\_intersect} = 1 - \frac{AVG_{queries}}{L_i} \tag{13}$$

Let $S_{not\_intersect}$ denote the search space of the APR-tree when the interval query Q does not intersect the boundary. As shown in Figure 14(a), since the interval query Q intersects only R_i and $L_i = \alpha \cdot AVG_{queries}$, the search space of the APR-tree is:

$$S_{not\_intersect} = \alpha \cdot AVG_{queries} \tag{14}$$

Let $S_{intersect}$ denote the search space of the APR-tree when the interval query Q intersects the boundary. As shown in Figure 14(b), since the interval query Q intersects both R_i and R_{i+1}, and $L_i = L_{i+1} = \alpha \cdot AVG_{queries}$, the search space of the APR-tree is:

$$S_{intersect} = 2\alpha \cdot AVG_{queries} \tag{15}$$

Let S_{avg} denote the average search space of the APR-tree for the two cases as shown in Figure 14. From Formulas (12), (13), (14), and (15), the average search space of the APR-tree is:

$$S_{avg} = P_{intersect} \cdot S_{intersect} + P_{not\_intersect} \cdot S_{not\_intersect} = (\alpha+1) \cdot AVG_{queries} \tag{16}$$

In Formula (16), since $\alpha \geq 1$, $\alpha = 1$ minimizes the average search space of the APR-tree. When $L_i = AVG_{queries}$, $S_{avg} = 2 \cdot AVG_{queries}$. Consequently, if $L_i \geq AVG_{queries}$, $L_i = AVG_{queries}$ minimizes the query cost. As a result, $L_i = AVG_{queries}$ reduces the number of R_i's which are searched to 2. and the average search space of the APR-tree is minimized. Therefore, $L_i = AVG_{queries}$ minimizes the average cost of the query Q whose time interval length is $AVG_{queries}$.

As shown in Formula (6), L_i determines the ratio of duplicate records. In other words, the smaller L_i, the larger size of the APR-tree. Figure 15 illustrates the size of the APR-tree as a function of L_i. In this figure, the difference between $Size(3DR-tree)$ and $Size(APR-tree)$ is due to duplicate records spawned by the boundary. To prevent that the size of the APR-tree becomes prohibitively large, L_i should have the minimum threshold. Therefore, we use $AVG_{records}$ as the minimum value of L_i. $L_i \geq AVG_{records}$ assures that the size of the APR-tree does not exceed 2 times that of the 3DR-tree since by Formula (6), when $L_i = AVG_{records}$, $\lambda_i \leq 0.5$. Namely, $\lambda_i = 0.5$ means that most of records intersect boundaries, so the number of duplicate records nearly equals the number of non-duplicate records.

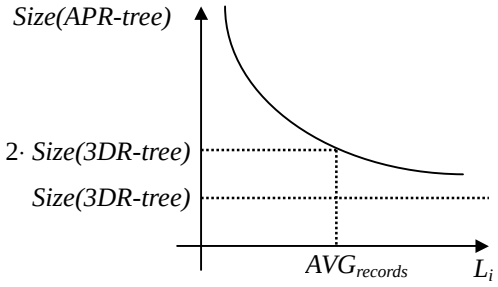


Fig. 15. The size of the APR-tree as a function of L_i

To determine L_i dynamically, we take advantage of query workloads. Figure 16 shows the algorithms to compute $AVG_{queries}$ and $AVG_{records}$. They are recomputed when queries and records are entered so that they can be dynamically adjusted. In Figure 16, $N_{queries}$ and $N_{records}$ denote the numbers of queries and records, respectively.

Using $AVG_{queries}$ and $AVG_{records}$ obtained from the query workload and the dataset, respectively, the tuning parameter L_i is dynamically determined by the following formula.

$$L_i = max(AVG_{queries}, AVG_{records}) \tag{17}$$

where $max(AVG_{queries}, AVG_{records})$ chooses the larger value between $AVG_{queries}$ and $AVG_{records}$.

Formula (17) guarantees the following two things: The first is that the size of the APR-tree does not exceed 2 times that of the 3DR-tree since $L_i \geq$

proc Compute_$AVG_{queries}$ (query $q =< S, [t_0, t_1), results >$)
begin
1. $AVG_{queries} := \frac{N_{queries} \times AVG_{queries} + (t_1 - t_0)}{N_{queries}+1}$
2. $N_{queries} := N_{queries} + 1$
end

proc Compute_$AVG_{records}$ (record $r =< id, s, [t_0, t_1) >$)
begin
1. $AVG_{records} := \frac{N_{records} \times AVG_{records} + (t_1 - t_0)}{N_{records}+1}$
2. $N_{records} := N_{records} + 1$
end

Fig. 16. Computation of $AVG_{queries}$ and $AVG_{records}$

$AVG_{records}$. The second is that when $AVG_{queries} \geq AVG_{records}$, the query cost is smallest since $L_i = AVG_{queries}$. However, when $AVG_{queries} < AVG_{records}$, the query cost is not smallest since $L_i > AVG_{queries}$ and the APR-tree suffers from the enlarged search space. However, the performance of the APR-tree would not degrade unless $AVG_{records}$ is much larger than $AVG_{queries}$.

5 Experiments

In this section, we compare the APR-tree with the HR-tree, the 3DR-tree and the MV3R-tree in terms of query cost, size, and update cost. Section 5.1 describes the experimental environment and Section 5.2 provides the experimental results.

5.1 Experimental Environment

Due to the lack of real data, synthetic datasets with real world semantics are generated by the GSTD method [13] which has been widely employed (e.g., [8,9,11,12]) as a benchmarking environment for access methods handling moving points and regions. Objects change their shapes or positions randomly. We investigate their movements for 500 timestamps. The cardinality of objects is 10,000. Objects are uniformly distributed in the spatial area. Agilities of datasets are 10%, 20%, and 30%, where the agility denotes the ratio of objects which change their positions or shapes at each timestamp. For example, agility = 10% means that 10% of the objects change their positions or shapes at each timestamp. To estimate $AVG_{queries}$, 10% of queries in the workload are randomly selected. $AVG_{records}$ is computed using records at the index building time.

Note that in this work, the R-tree implementations of all the spatio-temporal access methods are based on algorithms of the R*-tree [16]. The values of parameters of the MV3R-tree are the same as those in [11]: $P_{wv} = 0.35$ and $P_{svo} = 0.85$, where P_{wv} indicates the value for the weak version condition and P_{svo} indicates the value for the strong version overflow condition. A node corresponds to a page, whose size is 1 KBytes. Using this size, the maximum fanouts of the HR-tree, the 3DR-tree, the MV3R-tree, and the APR-tree are 50, 36, 36, and 36, respectively. In the HR-tree, the other nodes except root nodes do not keep temporal information, so the maximum fanout of the HR-tree is larger than those of the other access methods.

To simulate real life situations, the five query workloads in Table 3 are employed. Each workload has 1,000 queries. Spatial areas of queries are 1%, 3%, 5%, 7%, and 9% of the whole spatial area. The maximum time interval length of queries are between 1% ($5 = 500 \times 1\%$) and 20% ($100 = 500 \times 20\%$) of whole time interval.

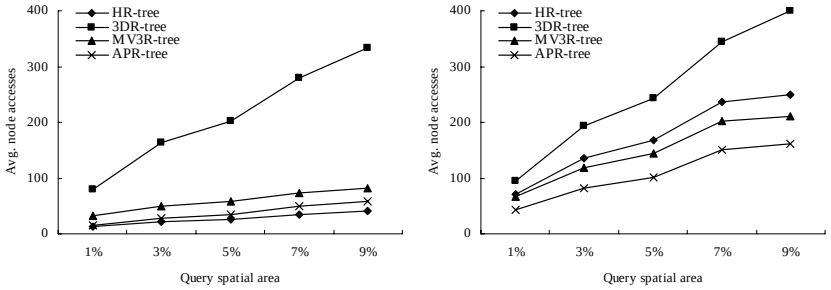
Table 3
Five workloads which consist of timestamp and interval queries

	Timestamp queries	**Interval queries**
The 1^{st} workload	100%	0%
The 2^{nd} workload	75%	25%
The 3^{rd} workload	50%	50%
The 4^{th} workload	25%	75%
The 5^{th} workload	0%	100%

To show the efficiency of the APR-tree, we measure the average number of nodes accessed to answer timestamp and interval queries in the workload. The justification for performance metric is that the smaller number of node accesses, the fewer disk I/O's needed to evaluate timestamp and interval queries.

5.2 Experimental Results

Figure 17 shows the performance of each access method for various query workloads when the dataset agility is 20%. Since the experimental results for dataset agilities = 10% and 30% are similar to those for the dataset agility = 20%, the experimental results for dataset agilities = 10% and 30% are omitted. For the workload with 0% interval queries (i.e., 100% timestamp queries), the 3DR-tree shows the worst performance, and the HR-tree shows the best performance since the search space of the HR-tree is optimal for timestamp queries and the average fanout of the HR-tree is much larger than those of the other access methods. Conversely, for the workload with 100% interval queries, the HR-tree shows the worst performance since the HR-tree should search many R-trees. The MV3R-tree outperforms the HR-tree and the 3DR-tree for



(a) 0% interval queries (b) 25% interval queries

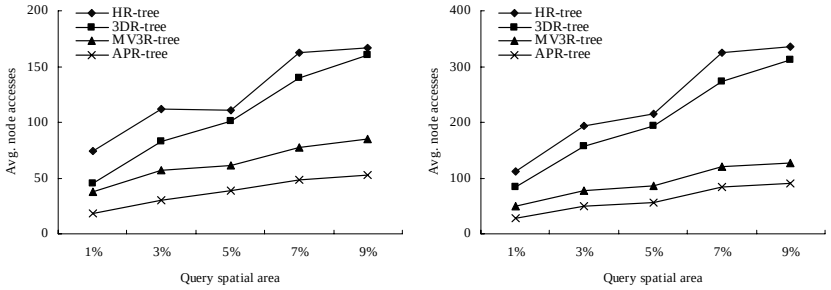
(c) 50% interval queries (d) 75% interval queries

(e) 100% interval queries

Fig. 17. Query costs for various query workloads (agility = 20%)

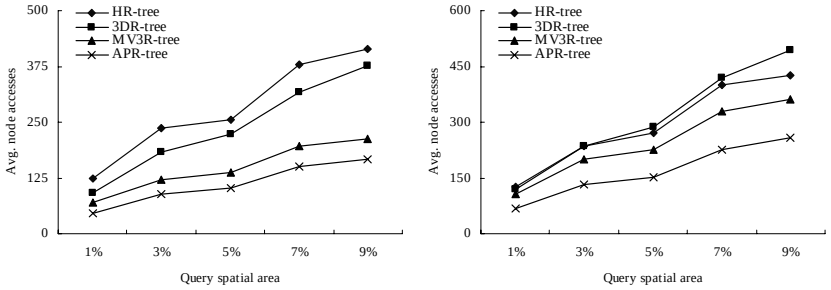
the workloads with 25%, 50%, and 75% interval queries since the MV3R-tree combines the MVR-tree for timestamp queries and the auxiliary 3DR-tree for interval queries. Thus, for the workload with 100% interval queries, the performance of the MV3R-tree is similar to that of the 3DR-tree since the auxiliary 3DR-tree also searches the whole history like the 3DR-tree. For the workloads with above 25% interval queries, the APR-tree achieves the best performance since $AVG_{queries} > AVG_{records}$ and $L_i = AVG_{queries}$. However, for the workload with 0% interval queries, the performance of the APR-tree is worse than that of the HR-tree since the search space of the APR-tree is larger than that of the HR-tree and the average fanout of the APR-tree is smaller than that of the HR-tree. However, as shown in Figures 17(b), 17(c), 17(d), and 17(e), the APR-tree outperforms the other access methods for a variety of query workloads.

Figure 18 shows the performance of each access method for the workload with



(a) Agility = 1%

(b) Agility = 5%



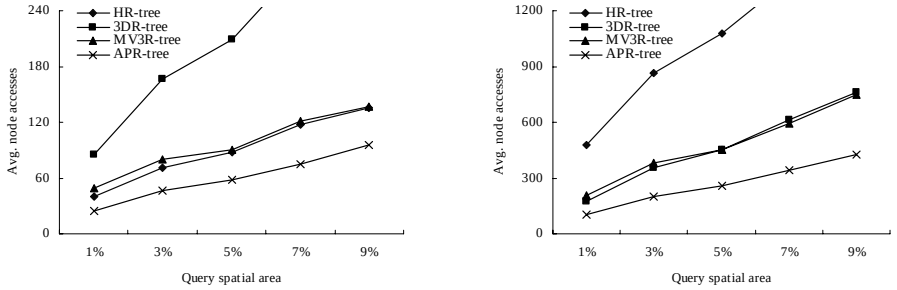
(c) Agility = 10%

(d) Agility = 30%

Fig. 18. Query costs for various agilities (50% interval queries)

50% interval queries when the dataset agility varies from 1% to 30%. Note that Figure 17(c) shows the result for the workload with 50% interval queries when dataset agility is 10%. Regardless of the dataset agility, the APR-tree accomplishes the best performance for this workload since $AVG_{queries} > AVG_{records}$ and $L_i = AVG_{queries}$. When the dataset agility is 30%, the APR-tree outperforms the other access methods. This is due to the fact that $AVG_{records}$ decreases as the agility increases. By Formula (6), the smaller $AVG_{records}$ results in the smaller number of duplicate records. By Formula (11), the smaller number of duplicate records improves the performance of the APR-tree since the total number of records decreases. Regardless of the dataset agility, the MV3R-tree also outperforms the HR-tree and the 3DR-tree since the MV3R-tree combines the MVR-tree responsible for timestamp queries and the auxiliary 3DR-tree responsible for interval queries. Since the number of duplicate data decreases as the agility increases, this improves the performance of the HR-tree. When the dataset agility is 10%, the 3DR-tree outperforms the HR-tree. In contrast, when the dataset agility is 30%, the HR-tree shows as similar performance as the 3DR-tree.

Figure 19 shows the performance of each access method for the workload with 100% interval queries when the dataset agility is 20% and maximum interval length of queries varies from 5 (= 500 × 1%) to 100 (= 500 × 20%). Note that Figure 17(e) shows the result for the workload with 100% interval queries when the dataset agility is 20% and maximum interval length of queries is 50 (= 500 × 10%). When maximum interval length is 5 as shown in Figure 19(a), the 3DR-tree shows the worst performance due to a large search space while



(a) Maximum length = 5 (1%) (b) Maximum length = 100 (20%)

Fig. 19. Query costs for various maximum interval lengths (agility = 20% and 100% interval queries)

the HR-tree and the MV3R-tree yield similar performance. In contrast, when maximum interval length is 100 as shown in Figure 19(b), the HR-tree shows the worst performance due to an access to a large number of R-trees while the 3DR-tree and the MV3R-tree yield similar performance. The APR-tree outperforms the others for various maximum interval length since it employs the query adaptive index scheme.

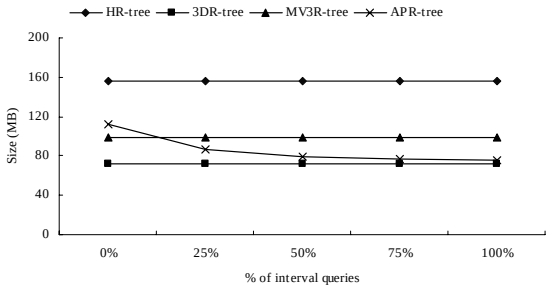


Fig. 20. The size comparison for various query workloads (agility = 20%)

Figure 20 shows the size of each access method for various query workloads when the dataset agility is 20%. The 3DR-tree has the smallest size since it has no duplicate data. The size of the HR-tree is about 2.7 times larger than that of the 3DR-tree since it has a huge number of duplicate data. The size of the MV3R-tree is about 1.5 times larger than that of 3DR-tree since it has a large number of duplicate data due to version splits. As shown in Figure 20, the sizes of the HR-tree, the 3DR-tree and the MV3R-tree are determined by datasets. In other words, the sizes of these access methods are not affected by query workloads. In contrast, the size of the APR-tree is associated with query workloads as well as datasets since L_i is determined by $AVG_{queries}$ and $AVG_{records}$. The size of the APR-tree is on the average 1.3 times larger than that of the 3DR-tree. As the ratio of timestamp queries increases, the size of the APR-tree increases due to the smaller L_i. For the workload with 0% interval queries (i.e., 100% timestamp queries), the size of the APR-tree is about 1.7 times larger than that of the 3DR-tree since $AVG_{queries} = 1$ is smaller than $AVG_{records}$ and by Formula (17), $L_i = AVG_{records}$ causes a large number of duplicate records. Although for the workload with 0% interval queries,

the size of the APR-tree is larger than that of the MV3R-tree, the APR-tree outperforms the MV3R-tree as shown in Figure 17(a). For the workloads with above 25% interval queries, the size of the APR-tree is smaller than that of the MV3R-tree. Especially, for the workload with 100% interval queries, the size of the APR-tree approaches to that of the 3DR-tree.

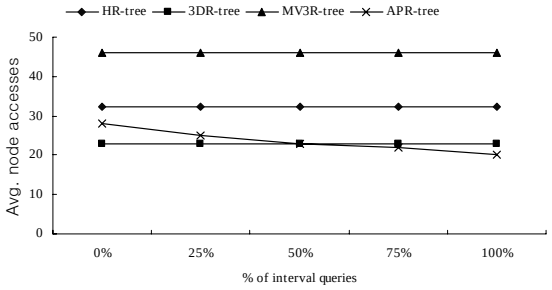


Fig. 21. Update costs for various query workloads (agility = 20%)

Figure 21 shows the update cost of each access method for various query workloads when the dataset agility is 20%. The update cost denotes the average number of nodes accessed to update each structure for object changes (insertion and logical deletion) and is measured by using the following equation: Update cost = $\frac{N_2}{N_1}$ where N_1 denotes the total number of records and N_2 denotes the total number of nodes accessed to insert N_1 records.

Compared with the HR-tree and the MV3R-tree, the update cost of the 3DR-tree is smallest since it has the smallest size and maintains the single structure. The update cost of the MV3R-tree is typically 2 times higher than that of the 3DR-tree since the MV3R-tree has to modify the MVR-tree and the auxiliary 3DR-tree concurrently. The update cost of the HR-tree is about 1.4 times higher than that of the 3DR-tree since the size of the HR-tree is large and nodes in the path between the leaf and the root node may be modified or created even when an object changes its shape or position. Since the size and the height of the APR-tree change according to query workloads, the update cost of the APR-tree is also affected by query workloads. Since the height of the APR-tree is lower than that of the 3DR-tree and the size of the APR-tree is on the average 1.3 times larger than that of the 3DR-tree, the update cost of the APR-tree is on the average similar to that of the 3DR-tree. As the ratio of interval queries increases, the size and the update cost of the APR-tree decrease since L_i increases.

6 Conclusions

To overcome the problems with the HR-tree, the 3DR-tree and the MV3R-tree for timestamp and interval queries, we proposed the APR-tree which takes

account of query workloads. Since the jurisdiction interval length of the APR-tree is automatically fitted to the query workload, the APR-tree outperforms the other access methods for various query workloads. Unlike the other access methods, the size and the update cost of the APR-tree are affected by query workloads. As the ratio of interval queries increases, the size and the update cost of the APR-tree decrease. The size of the APR-tree is on the average 1.3 times larger than that of the 3DR-tree. The update cost of the APR-tree is on the average similar to that of the 3DR-tree. Additionally, since the HR-tree and the MV3R-tree have the graph structure, when dealing with interval queries, they require additional overheads, such as the maintenance of the access list in order to avoid re-accessing nodes. However, since the 3DR-tree and the APR-tree have the tree structure, they do not require such overheads.

References

[1] G. Kollios, D. Gunopulos, and V. J. Tsotras, A. Delis, and M. Hadjielefteriou: Indexing Animated Objects Using Spatiotemporal Access Methods, TKDE 13(5), 2001.

[2] Y. Theodoridis and T. K. Sellis: A Model for the Prediction of R-tree Performance. ACM PODS, 1996.

[3] M. A. Nascimento and J. R. O. Silva: Towards historical R-trees. ACM SAC, 1998.

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

[5] P. K. Agarwal, L. Arge, and J. Erickson: Indexing Moving Points. ACM PODS, 2000.

[6] G. Kollios, D. Gunopulos, and V. J. Tsotras: On Indexing Mobile Objects. ACM PODS, 1999.

[7] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez: Indexing the Positions of Continuously Moving Objects. ACM SIGMOD, 2000.

[8] D. Pfoser, C. S. Jensen, and Y. Theodoridis: Novel Approaches in Query Processing for Moving Object Trajectories. VLDB, 2000.

[9] Y. Tao and D. Papadias: Efficient Historical R-Trees. SSDBM, 2001.

[10] M. Vazirgiannis, Y. Theodoridis, and T. K. Sellis: Spatio-Temporal Composition and Indexing for Large Multimedia Applications. Multimedia Systems 6(4), 1998.

[11] Y. Tao and D. Papadias: MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. VLDB, 2001.

[12] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis: Evaluation of Access Structures for Discretely Moving Points. Spatio-Temporal Database Management, 1999.

[13] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento: On the Generation of Spatiotemporal Datasets. SSD, 1999.

[14] I. Kamel and C. Faloutsos: On Packing R-trees. CIKM, 1993.

[15] A. Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. ACM SIGMOD, 1984.

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

[17] Y. Theodoridis, E. Stefanakis, and T. K. Sellis: Efficient Cost Models for Spatial Queries Using R-Trees. TKDE 12(1), 2000.