SEST$_L$: An Event-Oriented Spatio-Temporal Access Method*

Gilberto A. Gutiérrez
Universidad del Bío-Bío
Universidad de Chile
Blanco Encalada 2120
Santiago / Chile
ggutierr@dcc.uchile.cl

Gonzalo Navarro
Department of Computer Science
Universidad de Chile
Blanco Encalada 2120
Santiago / Chile
gnavarro@dcc.uchile.cl

Andrea Rodríguez
Department of Computer Science
Universidad de Concepción
Edmundo Larenas 215
Concepción / Chile
andrea@udec.cl

Abstract

This work presents a new access method (SEST$_L$) that handles discrete change events over objects' spatial attributes. The access method combines snapshots and events in *log* structures associated with space partitions. The definition of this new access method aims at extending capabilities of current spatio-temporal access methods to new types of queries (i.e., event-oriented queries), while competing with current structures for traditional *time-slice* and *time-interval* queries. The paper describes the structure and presents favorable analytical and experimental cost analysis of the structure.

1 Introduction

Spatio-temporal databases are composed of spatial objects that change their location or shape at different time instants [16]. Their objective is to model and represent the dynamic nature of real world applications [10]. Examples of these applications are transportation, monitoring, environmental, and multimedia systems.

*Gilberto Gutiérrez is partially funded by Dirección de Investigación, Universidad del Bío-Bío, Grant 043318 3/R. Gonzalo Navarro is funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile. Andrea Rodríguez is funded by Fondecyt 1050944, Conicyt, Chile.

Proceedings of the 32nd VLDB Conference, Seoul, Korea, 2006

Related to *window queries* in spatial databases, the most studied types of queries in spatio-temporal databases are *time-slice* and *time-interval* queries [14]. *Time-slice* queries retrieve all objects that intersect the query window at a particular time instant. *Time-interval* queries extend the idea of *time-slice* queries by considering consecutive time instants. All these queries focus on the coordinate- or snapshot-based representation of objects' movements. Recent studies, however, have emphasized the relevance of handling, as complementary information, snapshots and events, encouraging research in the integration of both types of information [20, 5, 2].

Our previous work [6] defines a preliminary spatio-temporal access method (SEST-Index) that handles not only *time-slice* and *time-interval* queries, but also event-based queries. In this paper, we present a new access method (SEST$_L$) that handles discrete changes over objects' spatial attributes using *log* structures associated with space partitions. This new access method improves and extends our previous results [6] by optimizing the use of *logs* in the data structure and by giving experimental and analytical comparisons with state-of-the-art spatio-temporal access methods. The work focuses on data about discrete events that result in changes of objects' locations; however, the results can be used in other application domains, such as applications that deal with changes in objects' shapes. Although in this paper we treat events and changes as synonyms, we point out some relevant literature addressing the conceptual differences among changes, events, and processes [20, 9, 4, 5].

The main contributions of this work are:

1. It presents a new spatio-temporal access method that handles snapshots and events associated with space partitions.

2. It compares the data structure against SEST-Index [6] and MVR-tree [13, 14]. To the best of our knowledge, SEST-Index is the only previ-

ous access method that addresses snapshots and events, and MVR-tree is a structure that outperforms previous spatio-temporal access methods in terms of time and space requirements.

3. It presents an analytical cost model of the structure that is compared with respect to experimentally validated.

The organization of the paper is as follows. Section 2 reviews current spatio-temporal access methods for applications that handle discrete changes. Section 3 describes the proposed access method in terms of its data structure and operations. Section 4 gives experimental evaluations with respect to SEST-Index and MVR-tree, which is followed by Section 5 with the presentation and evaluation of the cost model. Section 6 considers incorporating global snapshots into SEST$_L$. Conclusions and future work are given in Section 7.

2 Related Work

This section gives a classification of the most important spatio-temporal access methods for answering *time-slice* and *time-interval* queries. It focuses on the SEST-Index and MVR-tree structures, against which we compare the new proposed structure.

A classification of the previous spatio-temporal access methods is the following:

1. Methods that treat time as another dimension.
2. Methods that incorporate the temporal information in the node structure without considering time as another dimension.
3. Methods based on overlapping of the structure.
4. Methods based on multiversion of the structure.
5. Methods based on snapshots and events.

The 3D R-tree [19] considers time as another axis along with the spatial coordinates. In a tree-dimensional space, two line segments $\overline{[(x_i, y_i, t_i), (x_i, y_i, t_j))}$ and $\overline{[(x_j, y_j, t_j), (x_j, y_j, t_k))}$ model an object that initially remains at (x_i, y_i) during time interval $[t_i, t_j)$, and then, it locates at (x_j, y_j) during time interval $[t_j, t_k)$. Such line segments can be indexed by a 3D R-tree. This idea works well if all the final limits of the time intervals are known in advance. The 3D R-tree structure is efficient in space and in processing *time-interval* queries. It is, however, inefficient for processing *time-slice* queries.

RT-tree [21] is a structure of the second category where the temporal information is kept in the nodes of the R-tree. This is an extension of the information content of a traditional R-tree. The temporal information plays a secondary role because the search is guided by the spatial information. In this way, queries with temporal conditions cannot be efficiently processed [10].

HR-tree [11, 10] and MR-tree [21] are based on the concept of overlapping. The basic idea is that, given two trees, the most recent tree corresponds to an evolution of the older tree and subtrees can be shared between the older and newer trees. The major advantage of the HR-tree is its efficiency in processing *time-slice* queries. The major disadvantage is the excessive space that it requires to store the structure. For example, if only an object of each leaf node moves at instant t_i, the tree is completely duplicated at instant t_{i+1}.

MVR-tree [13, 12, 14] is a type of structure that handles multiversions. It is an extension of MVB-tree [1], where the time varying attribute is of a spatial type. Similar to the MVB-tree, each entry in the MVR-tree is of the form $\langle S, t_s, t_e, pointer \rangle$, where S corresponds to its MBR. An entry is *alive* at instant t if $t_s \leq t < t_e$.

MVR-tree imposes constraints on the number of entries stored in its nodes. A constraint ensures that there exist zero or at least $b \cdot p_{version}$ entries in any non-leaf node at a time instant t, where $p_{version}$ is a parameter of the tree and b is the capacity of a node. This condition groups *alive* entries at the same time instant for processing *time-slice* queries. Other constraints ensure a good use of space in the algorithms for insertion and deletion [13, 12, 14].

Figure 1 represents a 3D illustration of an example of the MVR-tree. In the example, extracted from [13], $b = 3$ and $p_{version} = \frac{1}{3}$. The objects A and G (thin lines) were inserted in alphabetic order. The example illustrates that when object C is inserted, node H becomes full so that a new node I is created. This new node I stores a copy of object C (note that C is the only object alive in node H). At the same time, the value of t_e changes to t in entry H, closing the time interval of the node. In Figure 1, nodes I, C and K are alive at the current time instant and D, E, G and F are dead nodes.

Like the MVB-tree, a MVR-tree has multiple R-trees (logical trees) that organize the spatial information for non-overlapping temporal windows. This structure outperforms the HR-tree in space and processing of short *time-interval* queries. As a modification of MVR-tree, MV3R-tree [13] improves the performance of the MVR-tree for long *time-interval* queries by adding an auxiliary 3D R-tree for processing those queries. Despite the improvement of MV3R-tree with respect to queries of long *time-intervals*, MV3R-tree violates a fundamental principle of multiversion structures, which establishes that dead nodes must not be modified.

A recent work [6] proposes an access method SEST-Index that belongs to the fifth category. A general scheme of SEST-Index is shown Figure 2. The idea of SEST-Index consists in maintaining the snapshots of the database for certain time instants (by using an R-tree) and a *log* to store the events occurred between

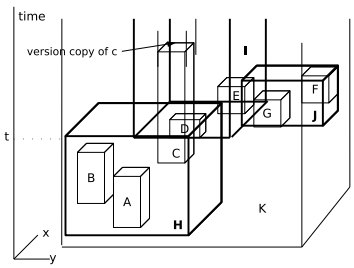


Figure 1: Example of a MVR-tree [13]

consecutive snapshots. The *log* is stored in time-order and allows us to reconstruct whatever the state of the database was between two consecutive snapshots. For example, in Figure 2 the state of the objects in snapshot t_0 are stored in R_0, and the events that modify the geometry of objects in the temporal interval (t_0, t_i), are stored in *log* L_0. Thus, to recover the state of the database at an instant t with $t_0 < t < t_i$, we start from the R-tree at instant t_0 and update objects' attributes (i.e., location) with the information of *log* L_0.

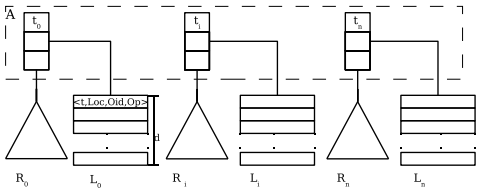


Figure 2: General outline of SEST-Index

3 Proposed Method: SEST$_L$

Similar to SEST-Index [6], the idea behind SEST$_L$ consists in maintaining snapshots for some time instants and storing the events that occur between consecutive snapshots. One of the main disadvantages of SEST-Index is the rapid growth of its size (storage use) as the number of changes increases. This disadvantage is explained because each snapshot duplicates all the objects, including those that have undergone no modification between consecutive snapshots. A solution to this problem was proposed in [6], but it has two important limitations: (1) the objects must be points and (2) the region where the changes occur must be fixed. SEST$_L$ overcomes these two limitations and maintains a similar performance.

Using a unique R-tree [7], SEST$_L$ splits the space into several regions at the leaf level and assigns a *log* to each of these regions. In SEST$_L$ a *log* is a structure that stores (leaf) snapshots and events. Figure 3 presents a region A and its corresponding *log* with three objects at instant t_0. At instant t_i, region A has grown to include a fourth object, situation that is reflected on the corresponding snapshot. The changes that occurred between t_0 and t_i are stored as events in the *log* associated with A.

In SEST$_L$, areas of both the regions to which the logs are assigned and the MBRs of non-leaf nodes in the R-tree are always growing along time. Due to this situation, the overlapping area of non-leaf nodes in the R-tree increases and the efficiency of query processing decreases. To overcome this situation, it is possible to define global snapshots (see Section 6), which creates new R-trees and therefore, new space partitions for each *log* at different time instants.

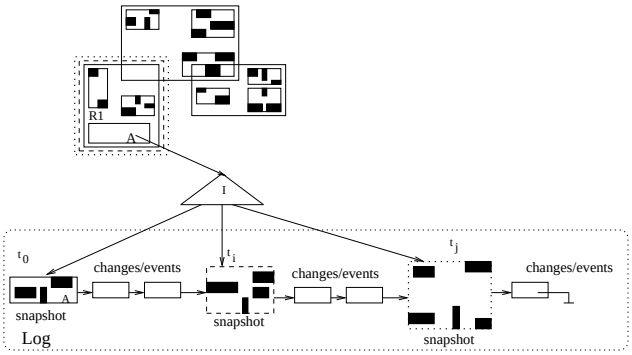


Figure 3: General outline of SEST$_L$

3.1 Structure description

The structure is an R-tree [7], where the leaves are *logs*. The *log* is a linked list of blocks. A *log* has two types of entries: event or change entries and snapshot entries (the first entry is always a snapshot). These entries in the *log* follow a temporal order.

An event entry is a tuple with the structure $\langle t, Geometry, Oid, Op \rangle$, where t corresponds to the time when the change occurred, and Oid is the object identifier. $Geometry$ corresponds to the spatial component of the object, which depends of the geometric type (i.e., point, line, polygon or MBR) and dimension (2D or 3D). Finally, Op indicates the type of operation (i.e., type of event or change).

This work considers only two types of events: *move_in* (i.e, an object moves to a new location) and *move_out* (i.e., an object leaves its current location). Thus an object creation is modeled as a *move_in*, an object deletion is modeled as a *move_out*, and an object movement is modeled as a *move_out* followed by a *move_in*. One could think in storing just one move event instead of two events for the change of location of objects. This decreases the storage cost of the structure, at the price of increasing the time cost for processing event-based queries. Later, in Section 4.4, we will analyze how much space can be saved if the structure is designed to support only *time-slice* and *time-interval* queries.

The second type of entries, a snapshot entry, stores the snapshot of a leaf node, with one entry for each object alive at the time instant when the snapshot is created.

Similar to SEST-Index [6], SEST$_L$ also considers a parameter d, equal for all logs in the structure, which defines the maximum number of events that are possible to store between consecutive snapshots.

3.2 Operations

3.2.1 *time-slice* queries

In order to process a *time-slice* query (Q, t), the first step is to find all leaves that intersect the query window (rectangle Q). Next, for the *log* of each leaf, the process obtains the corresponding snapshot according to the time instant t of the query. This snapshot is the one built for the latest time instant tr such that $tr \leq t$. The spatial objects stored in the selected snapshot which intersect the query window form an initial answer. Finally, this answer is updated with the changes stored in the event entries of the *log* within the time interval $(tr, t]$ (Algorithm 1). The whole answer is the union of those computed for each involved leaf.

Algorithm 1 Algorithm to process a *time-slice* query

```
1: time-sliceQuery(Rectangle Q, Time t, R-tree R)
2: let B = SearchRtree(Q, R). {B is the set of leaves (logs)
   that intersect Q}
3: G = ∅ {G is the set of objects that belong to the answer}
4: for each log b ∈ B do
5:    let tr be the time of the latest snapshot in b such that
      tr ≤ t.
6:    let A be the set of all objects alive in the snapshot created
      at the time instant tr in the log b.
7:    for each event entry c ∈ b such that tr < c.t ≤ t do
8:       if c.Geometry intersects Q then
9:          if c.Op = move_in then
10:            A = A ∪ {c.Oid}
11:         else
12:            A = A − {c.Oid}
13:         end if
14:      end if
15:   end for
16:   G = G ∪ A
17: end for
18: return G
```

3.2.2 *Time-Interval* queries

Processing *time-interval* queries, $(Q, [t_i, t_f])$ requires to find the set of spatial objects that intersect the query window (Q) at the initial instant t_i. This is equivalent to a *time-slice* query at instant t_i. Then, objects are updated based on the changes occurred within the interval $(t_i, t_f]$ (Algorithm 2).

3.2.3 Event queries

One of the novelties of the SEST$_L$ structure is its capability for processing not only *time-slice* and *time-interval*, but also event queries. For example, given a region Q and an instant t, an event query may be to find the number of objects that moved in or out from region Q at instant t. These types of queries are possible and useful in applications that aim to analyze the pattern of objects' movements [20, 5].

Processing event queries with SEST$_L$ (see Algorithm 3) is simple and efficient, since the structure explicitly stores the changes over objects' geometries. Algorithms for these types of queries are similar to those for *time-slice* and *time-interval* queries.

Algorithm 2 Algorithm to process a *time-interval* query

```
1: IntervalQuery(Rectangle Q, Time t_i, t_f, R-tree R)
2: let B = SearchRtree(Q, R). {B is the set of leaves (logs)
   that intersect Q}
3: G = ∅ {G is the set of objects that belong to the answer}
4: for each log b ∈ B do
5:    let tr be the time of the latest snapshot in b such that
      tr ≤ t.
6:    let A be the set of all objects alive in the snapshot created
      at the time instant tr in log b.
7:    update A with the changes stored in b occurred between
      (tr, t_i]. {like a time-slice query}
8:    ts = Next(t_i) {Next(x) returns the next instant after x
      when changes have been stored in log b}
9:    while t_s ≤ t_f ∧ there exist event entries in log b do
10:      for each event entry c ∈ b such that t_s = c.t do
11:         if c.Geometry intersects Q then
12:            if c.Op = move_in then
13:               A = A ∪ {c.Oid}
14:            else
15:               A = A − {c.Oid}
16:            end if
17:         end if
18:      end for
19:   end while
20:   G = G ∪ A
21:   ts = Next(ts)
22: end for
23: return G
```

3.2.4 Updating the structure

These operations update the structure upon changes that occur in each time instant. Let us assume that changes are stored in a list. When an object moves, two events, *move_out* and *move_in*, are created. Event *move_in* include all attribute values t, *Geometry* and *Oid* at the incoming object. Event *move_out*, in contrast, only contains the attribute values t and *Oid*. A simple way to know the value of *Geometry* is to keep a hash table ($\langle Oid, Geometry \rangle$) with the last *Geometry* value for each object. Another alternative is also use to a hash table, with a reference to the current *log* where the object was located just before the *move_out* event. For each *move_in* event, we choose the corresponding *log* where it should be inserted according to the clasical R-tree insertion policy [7] (called *chooseleaf()* in the pseudocode). *move_out* events are slightly more complicated because we have to ensure they are recorded in the same log of their corresponding *move_in* event.

Algorithm 3 Algorithm to process an event query

```
1: EventQuery(Rectangle Q, Time t, R-tree R)
2: let B = SearchRtree(Q, R). {B is the set of regions that intersect Q}
3: oi = 0 {number of objects that moved in to Q at instant t}
4: oo = 0 {number of objects that moved out from Q at instant t}
5: for each log b ∈ B do
6:    let tr be the time of the latest snapshot in b such that tr ≤ t.
7:    find the first event entry c ∈ b such that c.t = t.
8:    while c.t = t do
9:       if c.Geometry intersects Q then
10:         if c.Op = move_in then
11:            oi = oi + 1
12:         else
13:            oo = oo + 1
14:         end if
15:      end if
16:      c = NextChange(c) {NextChange(x) returns the event entry following x in log b}
17:   end while
18: end for
19: return (oi, oo)
```

Algorithm 4 Algorithm to update the structure

```
1: InsertChanges(Time t, Changes C, R-tree R, Integer d) {t is the time instant when a change or event occurs, C is the list of changes at t, and d is the capacity of a log to store event entries between snapshots}
2: for each c ∈ C do
3:    if c.Op = move_in then
4:       b = chooseleaf(R, c.Geometry)
5:    else
6:       b = chooseleafA(R, c.Geometry).
7:    end if
8:    let L be the list of events occurred in b after the last snapshot.
9:    let l be the number of changes stored in L.
10:   if l > d ∨ c is the first event entry that is inserted in b then
11:      create a new snapshot S
12:      create a new list L and assign it to S
13:   end if
14:   insert c at the end of L
15:   if c.Op = move_in then
16:      update the MBRs of all event entries in the path followed by chooseleaf() to reach b.
17:   end if
18: end for
```

As the R-tree shape may have changed since the time of the *move_in* event, *chooseleaf()* is not guaranteed to find the same *log* again. The solution is to carry out a spatial search for the *Geometry* corresponding to the *Oid* of the *move_out* event. This may involve following several paths in the R-tree. Note that this is complication does not arise if we record the log where each *Oid* is located (second hashing scheme discussed above). We call *chooseleafA()* this procedure in the pseudocode. In both cases, the new change is inserted as an event entry after the last snapshot that was stored in the corresponding *log*. If, at the instant of insertion, the number of changes exceeds a parameter d, a new snapshot is created (Algorithm 4). The insertion of a *move_in* event may require updates to the MBRs of the leaf as well as ancestor nodes whose MBRs must now include the *Geometry* of the arriving object.

4 Experimental Evaluation

A first experimental evaluation compares SEST$_L$ with SEST-index under the same conditions (i.e., data set and parameters) used in [6]. It considers that both structures use the minimum value for parameter d, when d is capable of storing all changes occurred at a time instant. This value of parameter d produces the best results in terms of efficiency to answer *time-slice* or *time-interval* queries given a known change frequency (percentage of objects that change from an instant to the next). In all cases, SEST$_L$ outperforms SEST-Index, not only in storage cost, but also in time cost. In the evaluations, SEST$_L$ uses between 15% and 50% of the storage used by SEST-Index, and the time to process a query was reduced to values between 20% and 46% of the time needed by SEST-Index. The unfavourable performance of SEST-Index with respect to SEST$_L$ can be explained because SEST-Index duplicates in each snapshot all objects, even those objects that have not moved, and because the logs in SEST-Index group changes by time and not by time and space.

Subsequently, a second experimental evaluation compares SEST$_L$ against MVR-tree under several scenarios. MVR-tree is considered to be the best alternative structure for answering *time-slice* and *time-interval* queries [13, 14, 12], outperforming the previous well-known HR-tree structure [11]. We tested the structures in terms of storage and time costs for for a database with 23,268 objects (points) that move along 200 time instants, and different types of queries. The change frequencies were 1%, 5%, 10%, 15%, 20% and 25%. We tested SEST$_L$ with values for parameter d of 2, 4 and 8 disk blocks, with a block size of 1024 bytes. Storage cost was measured by the number of blocks needed after inserting the objects and their changes. Access time (i.e., time cost) was defined as the average number of blocks read for performing 100 random queries.

The dataset was obtained with the spatio-temporal data generator GSTD [17] following a uniform distribution. We used the cost model for MVR-tree [14] to obtain the storage cost and the time cost for the queries. For SEST$_L$, in contrast, we used an experimental evaluation with the same input and variables.

4.1 Storage cost

Figure 4 shows that SEST$_L$ uses less disk space than MVR-tree, and this behavior becomes clearer as the change frequency increases. SEST$_L$ requires only 62% to 75% of the space needed by MVR-tree. We observed that, as the value of d increases, SEST$_L$ needs less storage than MVR-tree, which is explained by the

lower frequency of snapshots.

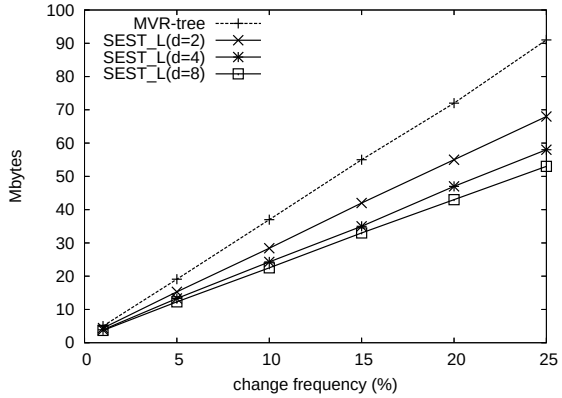


Figure 4: Space usage

4.2 *time-slice* and *time-interval* queries

Figure 5 shows that, for queries with time interval greater than 30 time units and query window formed by 6% of the dimension in each axis, SEST_L shows a better performance than MVR-tree for all number of blocks used to store events (i.e., 2, 4 and 8 blocks for the parameter d); however, when the time interval is less than 10 units, MVR-tree outperforms SEST_L.

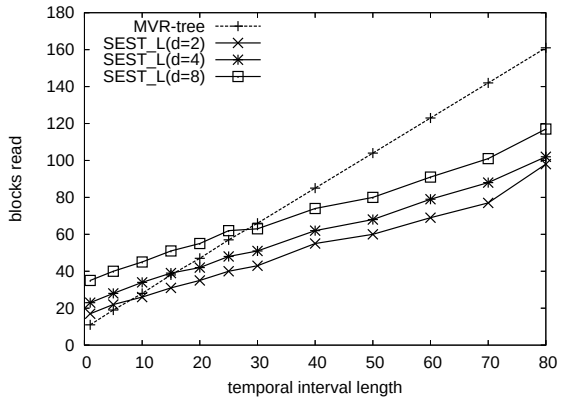


Figure 5: Blocks read by queries, 10% change frequency, query window formed by 6% of the dimension in each axis

Figures 6 and 7 show the behavior of both MVR-tree and SEST_L with *time-interval* queries using two fixed time-interval lengths (i.e. 10 and 40 units), and with a varying query window of 2%, 4%, 6%, 8%, 10% and 12% of the space in each dimension. Figure 6 reflects a similar time cost for SEST_L with respect MVR-tree, when the value of parameter d is equal to 2 blocks. The storage used by SEST_L, however, is less than the storage used by the MVR-tree for change frequency of 10% (Figure 4). SEST_L outperforms MVR-tree for all values of parameter d analyzed when the length of the time interval of the query is 40 time units (Figure 7).

These results indicate that, for *time-interval* queries over 30 time instants, SEST_L has better performance than MVR-tree. For *time-slice* queries, that is, *time-interval* queries with interval length equal to 1 unit, MVR-tree should overcome SEST_L.

SEST_L pays an important initial cost at retrieving objects until the initial instant of the query's time interval, which includes traversing the R-tree and processing the changes until such instant. Thereafter, the total time cost of the query is low, since it only requires to apply the changes until the final instant of the query's time interval. In the case of the MVR-tree, the portion of the R-tree that needs to be checked for a query increases with the length of the query's time interval. This explains why MVR-tree outperforms SEST_L in short intervals and SEST_L outperforms MVR-tree otherwise. Figure 8 shows the area where each structure dominates.

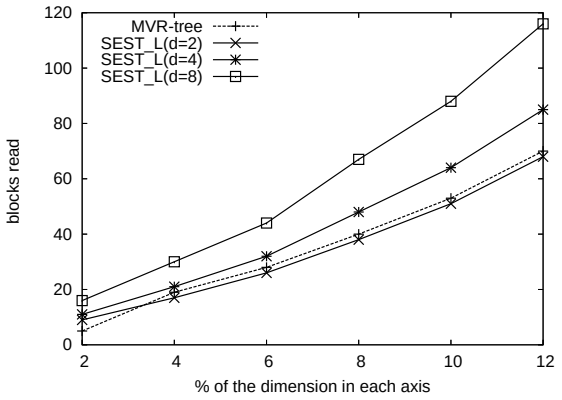


Figure 6: Blocks read by queries for change frequency of 10% and length of time interval of 10

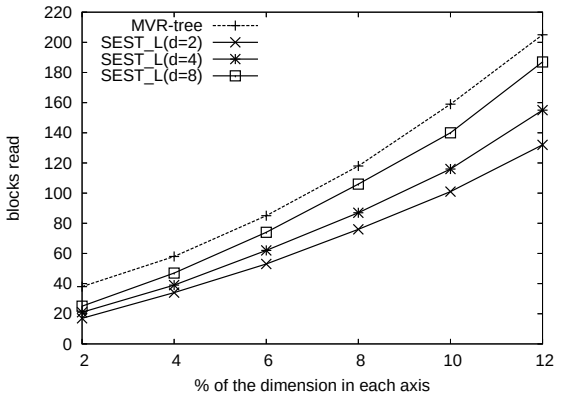


Figure 7: Blocks read by queries for change frequency of 10% and length of time interval of 40

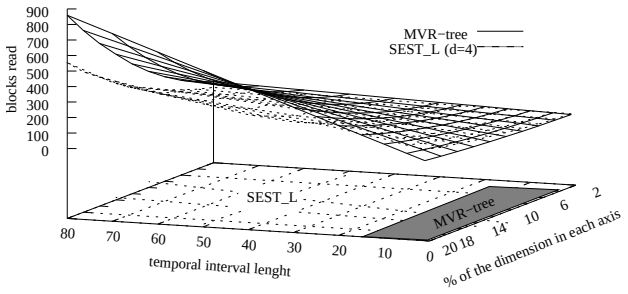


Figure 8: Blocks read by queries

4.3 Queries about events

As we explained in Section 3.2.3, SEST$_L$ allows answering event queries that occur at particular time instants. The cost of processing such types of queries is the same than of processing *time-slice* or *time-interval* queries. MVR-tree, in contrast, is an access method oriented to answer *time-slice* and *time-interval* queries and, therefore, it does not provide algorithms for event queries. Actually the current structure of the MVR-tree does not allow the processing of event queries, since the constraints imposed on the nodes of the tree (weak version, overflow and underflow), and the algorithms for maintaining the structure, make it possible to have entries with time intervals created artificially. For example, in Figure 1 an entry for object C in node H indicates an elimination time t with $t = t_e$, even when the object's location has not changed. Likewise, an entry in node I indicates an insertion time t, with $t_s = t$. Both entries are created with the purpose of keeping the condition of the weak version [13]. The existence of these artificial intervals makes it impossible to process event queries with the MVR-tree.

4.4 Adjusting the SEST$_L$ structure

SEST$_L$ is an event-oriented access method that aims to efficiently answer not only events, but also *time-slice* and *time-interval* queries. To fulfill the performance requirements for all queries, the structure maintains some data that can be eliminated if only *time-slice* or *time-interval* are of interest. In particular, the experimental evaluations considers the structure proposed in section 3.1, where a movement or change of location is represented by two events: *move_out* and *move_in*. In this section, we optimize the structure when only *time-slice* or *time-interval* queries are processed.

The optimization is a simple modification in the structure by eliminating the attribute *Geometry* in the *move_out* event. Figure 9 shows 30% of space savings by applying the modification of the structure. Likewise, this modification produces time savings for query processing as there is less data to read from disk (Figure 10). This indicates that, by adjusting SEST$_L$ to process only *time-slice* or *time-interval* queries, the advantages of SEST$_L$ increase with respect to storage and time requeriments.

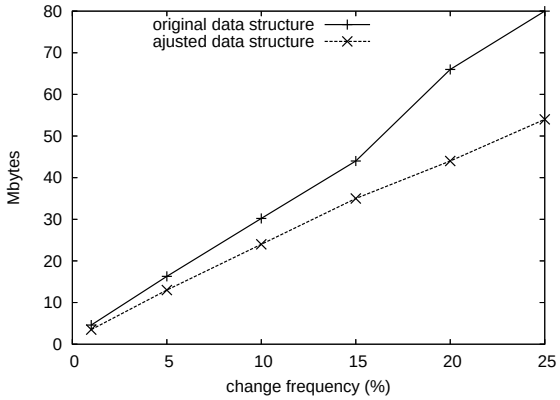


Figure 9: Space savings of SEST$_L$ (23,268 points, 200 time instants and $d = 2$)

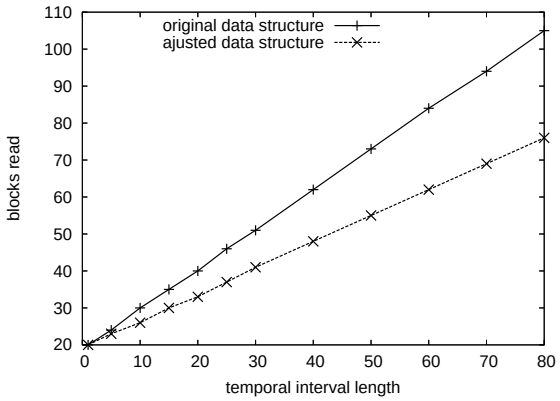


Figure 10: Time savings of ajusted SEST$_L$ (23,268 points, 200 time instants, 10% change frequency, $d = 4$ and 6% in each dimension for the query window)

5 A cost model for SEST$_L$

This section presents a cost model for SEST$_L$, which allows us to predict its storage and time costs for spatio-temporal queries. The cost model is compared with the actual experimental behavior to demostrate its prediction capability. Finally, SEST$_L$ and MVR-tree are compared using their corresponding cost models [14].

The cost model of SEST$_L$ assumes that the initial locations of moving objects and the subsequent objects' movements distribute uniformly. Figure 11 describes the variables used in the cost model of SEST$_L$.

5.1 Storage cost of the R-tree

Let N be the number of objects stored in an R-tree with fanout f. The height h of an R-tree is given by

Symbol	Description
ai	Width of the time interval of a query
c	Changes per block
D	Initial density of the set of objects
DA	Average number of nodes in an R-tree accessed for a spatial query
f	Average capacity of a node in an R-tree (fanout)
h	Height of an R-tree
il	Number of times instants that can be stored in a *log* between consecutive snapshots
l	Total number of changes stored between consecutive snapshots
N	Total number of objects
nl	Number of changes stored in a *log* adjusted to an integer number of time instants
nt	Time instants stored in the structure
p	Change percentage between time instants (change frequency)
$q = (q_1, \ldots, q_n)$	Query rectangle
TN	Total number of blocks used by an R-tree
NB	Number of *logs*
DA_{ts}	Number of blocks accessed for a *time-slice* query
DA_{in}	Number of blocks accessed for a *time-interval* query
TB_{total}	Total number of blocks needed to store the SEST$_L$ structure

Figure 11: Definition of variables

the equation $h = 1 + \left\lceil \log_f \frac{N}{f} \right\rceil$ [3].

Since the number of entries in a node is approximately f, it is possible to assume that the number of leaf nodes is $N_1 = \left\lceil \frac{N}{f} \right\rceil$ and that the number of non-leaf nodes at the level immediately superior to the leaves is $N_2 = \left\lceil \frac{N_1}{f} \right\rceil$. Considering that the level h is at the root and that the level 1 is at the leaves, the average number of nodes at level j is given by equation $N_j = \left\lceil \frac{N}{f^j} \right\rceil$ [15]. Thus the average number of nodes used (i.e., storage cost) for an R-tree is determined by Eq.(1).

$$TN = \sum_{j=1}^{h} \left\lceil \frac{N}{f^j} \right\rceil \approx h + \left\lfloor N \cdot \frac{(f^h - 1)}{f^h \cdot (f-1)} \right\rfloor \quad (1)$$

5.2 Time cost of the R-tree

Based on [18, 15], the number of nodes accessed (i.e., time cost) by an R-tree in window queries with n-dimensional spatial objects (DA_n) is given by Eq.(2) (see next for D_j).

$$DA_n = 1 + \sum_{j=1}^{h} \left\{ \frac{N}{f^j} . \prod_{i=1}^{n} \left(\left(D_j . \frac{f^j}{N} \right)^{\frac{1}{n}} + q_i \right) \right\} \quad (2)$$

Given that our experiments consider 2-dimensional objects, our DA is defined by Eq.(3), with $n = 2$ and assuming a square query window, $q = q_1 = q_2$.

$$DA = 1 + \sum_{j=1}^{h} \left\{ \left(\sqrt{D_j} + q \cdot \sqrt{\frac{N}{f^j}} \right)^2 \right\} \quad (3)$$

In Eq.(3), D_j corresponds to the density of spatial objects at level j [18, 15], which is obtained by Eq.(4) with D_0 being the density of spatial objects that need to be indexed in the R-tree[1].

$$D_j = \left(1 + \frac{\sqrt{D_{j-1}} - 1}{\sqrt{f}} \right)^2 , 1 \leq j \leq h \quad (4)$$

An important property of the Eqs.(2) and (3) is that they depend only on f, N, q and D_0 and, therefore, there is no need to construct the R-tree to estimate the performance of the query.

5.3 Storage cost of SEST$_L$

Two data describe the storage cost (number of blocks) used by the SEST$_L$: the number of *logs* and the space needed per *log*. The number of *logs* is equal to the number of leaves in the R-tree, which is expressed by equation $NB = \left\lceil \frac{N}{f} \right\rceil$.

The number of time instants (il) that are possible to store between snapshots is determined by Eq.(5).

$$il = \left\lceil \frac{l}{p \cdot f} \right\rceil \quad (5)$$

Using Eq.(5), it is possible to calculate the value nl (Eq.(6)) such that all events occurred at the same time instant are stored between the same snapshots.

$$nl = p \cdot f \cdot il \quad (6)$$

With il and nl, the number of blocks used for each *log* is given by Eq.(7). In this equation, the first term of the sum corresponds to the number of blocks that store all snapshots in each *log*, assuming that the live objects fit, on average, in a disk block. The second term represents the number of blocks required, on average, for each *log* to store all changes or events occurred in all time instants. The last term is used to obtain the number of blocks occupied for storing the changes or events occurred after the last snapshot.

$$TB_{log} = \left\lceil \frac{nt}{il} \right\rceil + \left\lfloor \frac{nt}{il} \right\rfloor \cdot \left\lceil \frac{nl}{c} \right\rceil + \left\lceil \frac{nt - \lfloor \frac{nt}{il} \rfloor \cdot il}{il} \cdot \frac{nl}{c} \right\rceil \quad (7)$$

Finally, the number of blocks used by the SEST$_L$ is given by Eq.(8).

$$TB_{total} = (TN - NB) + NB \cdot TB_{log} \quad (8)$$

[1] In this paper we have used $D_0 = 0$, as we index points

5.4 Time cost of SEST$_L$

The time cost of the SEST$_L$ structure can be estimated by adding the time cost of accessing the R-tree without leaves and the time cost of processing all logs that intersect the query window. In the following, Rp-tree refers to the R-tree without leaf nodes. The leaf nodes of the Rp-tree contain the MBRs for each of the leaf nodes in the R-tree.

The number of objects handled by the Rp-tree is $N_p = \frac{N}{f}$. Let $h_p = 1 + \left\lceil \log_f \frac{N_p}{f} \right\rceil$ be the height of the Rp-tree. The number of nodes accessed in the tree is given by Eq.(9).

$$DA_{Rp-tree} = 1 + \sum_{j=1}^{h_p} \left\{ \left(\sqrt{D_j} + q \cdot \sqrt{\frac{N_p}{f^j}} \right)^2 \right\} \quad (9)$$

For D_j we use Eq.(4), where D_0 is the density of leaves (i.e., D_1 in the original R-tree). The number of logs to be processed in a query corresponds to the density formed by the leaves' MBR and query's MBR. This number of logs is defined in Eq.(10), where $D_1 = \left(1 + \frac{\sqrt{D_0}-1}{\sqrt{f}}\right)^2$.

$$NL = 1 + \left(\sqrt{D_1} + q \cdot \sqrt{\frac{N}{f}} \right)^2 \quad (10)$$

Therefore, the number of blocks accessed in a *time-slice* query is defined by Eq.(11).

$$DA_{ts} = DA_{Rp-tree} + NL \cdot \left(1 + \left\lceil \frac{nl}{2 \cdot c} \right\rceil \right) \quad (11)$$

Likewise, the average number of blocks accessed in a *time-interval* query is given by Eq.(12).

$$DA_{in} = DA_{ts} + NL \cdot \left\lceil \frac{(ai-1) \cdot p \cdot f}{c} \right\rceil \quad (12)$$

5.5 Experimental evaluation of the cost model

In order to evaluate the cost model, new experimental evaluations were conducted with synthetic data obtained from GSTD [17]. These experiments use 23,268 objects (points) and 200 time instants with change frequencies of 1%, 5%, 10%, 15%, 20% and 25%. They also consider values of parameter d equal to 2, 4 and 8 blocks. The value f of the R-tree was set to 34 (68% of the capacity [15] of a node in an R*-tree that is able to hold 50 entries). Figures 12, 13 and 14 show the prediction capability of the cost model for both storage and time requeriments. Figure 12 indicates that the storage cost predicted by the model and the one obtained with the experiments are similar for all values of d analyzed, with a relative average error of 15%.

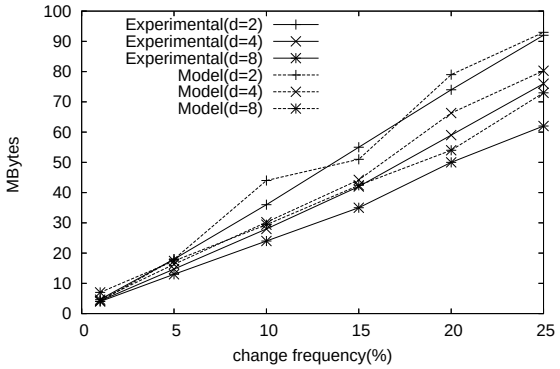


Figure 12: Estimation of the storage usage

Figures 13 and 14 show that the prediction in the time cost of query processing is very good, with a relative average error of 8%.

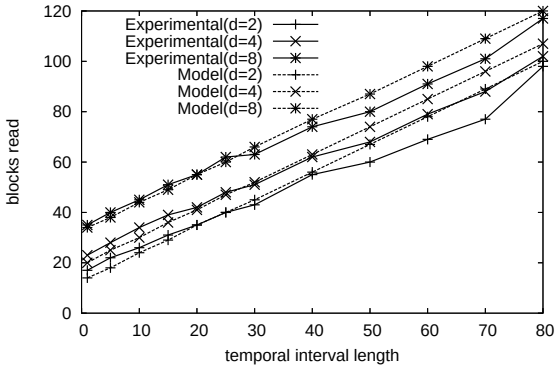


Figure 13: Estimation of the time cost of queries (10% change frequency and 6% in each dimension for query window)

5.6 Comparing MVR-tree and SEST$_L$ in new scenarios

The cost models for MVR-tree [14] and SEST$_L$ were compared by using the same number of objects (23,268) and snapshots (200) used in Section 5.5. Figure 15 shows the number of blocks accessed for a query that considers 12% of size along each dimension of the whole space for the query window. The parameter d was set to 4. Figure 15 indicates that SEST$_L$ overcomes MVR-tree from a length of the query time interval that is greater than 30 units. The advantage of SEST$_L$ over MVR-tree is larger when the change frequency increases. We can see in Figure 15 that when the change frequency is around 40% and the time interval is superior to 60 units of time, SEST$_L$ requires to access only 60% of the number of nodes accessed by MVR-tree.

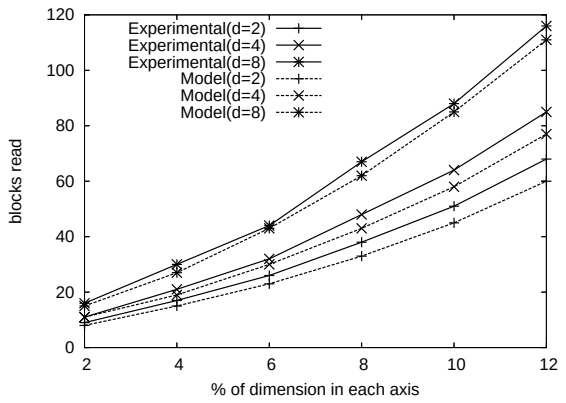


Figure 14: Estimation of the time cost of queries (10% change frequency and length of time interval equal to 10)

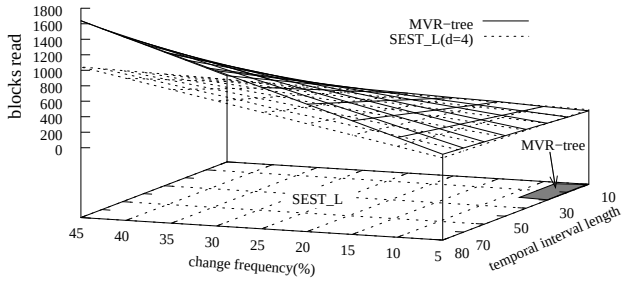


Figure 15: Performance of a query for different change frequencies and lengths of the query time interval

6 SEST$_L$ with global snapshots

A potential problem of SEST$_L$ is that the query performance can deteriorate as the size of the logs increase along time. When the size of the logs and, therefore, the density of the objects stored in the leaves of the Rp-tree (D_1 in Section 5.4) increase, the number of logs that are accessed to answer a query of later time instants is larger than the number of logs for queries at earlier time instants.

The value of the density ($realDen$) is defined by Eq.(13), where M is the set of all MBRs that are located at level 1 of the R-tree (i.e., at the leaves of the Rp-tree), and $TotalArea$ is the total area used by the objects.

$$realDen = \frac{\sum_{i \in M} Area_i}{TotalArea} \quad (13)$$

Figure 17 (*(1) Uniform distribution*) shows the growth of $realDen$ for 23,268 objects with uniform distribution until reaching an approximated value of 1. When the initial distribution is not uniform (Figure 16), however, $realDen$ is larger than the value for a uniform distribution (Figure 17). This degrades the performance of queries over time.

A solution to this problem is to define SEST$_L$ with global snapshots. A global snapshot is an R-tree that considers the position of all objects at a particular time instant. Such R-tree allows us to redefine the subregions associated with logs and, therefore, to decrease the density value ($realDen$) (Eq.(13)).

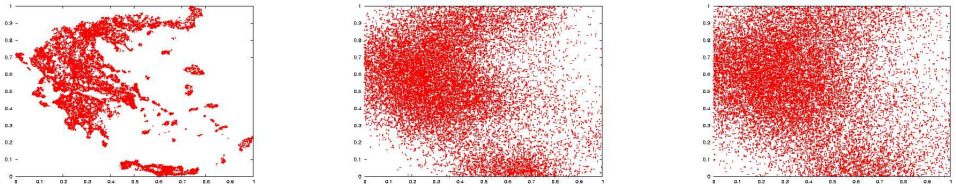
We propose to use the density value $realDen$ to determine the time instant for creating a new global snapshot. The process of creating a new global snapshot is described in Algorithm 5. This algorithm checks that $realDen$ after inserting new change events does not exceed the threshold $(1 + ls) \cdot lastDensity$, where $lastDensity$ is the density of the last global snapshot. When $realDen$ exceeds the threshold, a new R-tree is created. Let $newDensity$ be the density of the R-tree just created. We ensure that $newDensity < li \cdot realDen$ before actually creating the new R-tree, otherwise the improvment is not worth the extra space. Here $0 < ls, li < 1$ are parameters (see Algorithm 5).

Algorithm 5 Algorithm that controls the creation of global snapshots

```
1: newSnapShot(R-tree R, float lastDensity, Changes C, float ls, float li, Integer d, Time t) { R is the last
   R-tree created in SEST_L, lastDensity corresponds to the value of realDen for R when the last global snapshot was
   create, C is a list with changes to insert in SEST_L that occurred in the last time instant t, ls and li are fractions of
   lastDensity }
2: InsertChanges(t, C, R, d) {Inset changes in SEST_L using Algorithm 4}
3: Let newDensity be the new value of realDen for R after inserting the changes
4: if newDensity > (1 + ls) · lastDensity then
5:    Let newR-tree be the new R-tree created with the positions of objects at instant t.
6:    Let tmpDensity the value of realDen for newR-tree
7:    if tmpDensity < li · newDensity then
8:       R = newR-tree
9:       lastDensity = tmpDensity
10:   end if
11: end if
12: return (R, lastDensity)
```

6.1 Evaluation of SEST$_L$ with global snapshots

This section compares SEST$_L$ with global snapshots against MVR-tree by using the same objects (23,268 point objects and 10% of change frequency) that were considered in the experimental evaluation of the cost model of MVR-tree [14]. In this experiment, objects have a non-uniform initial distribution and move the next 199 time instants until reaching the final distribution described in Figure 16. We refer to this set of objects as NUD. The storage and time costs of MVR-tree were directly obtained from the data published in [14], whereas the storage and time costs of the SEST$_L$ were experimentally obtained with our implementation, considering 4 blocks for the parameter d.



(a) (b) (c)

Figure 16: Evolution of moving objects: (a) instant 0, (b) instant 100 and (c) instant 200

Figure 17 shows the values of density *realDen* under five different scenarios: (1) the density obtained with SEST$_L$, 23,268 objects (points), and an initial uniform distribution, (2) the density of SEST$_L$ with the set of objects *NUD* without global snapshots, (3), (4) and (5) densities of SEST$_L$ when considering the set of objects *NUD* and the thresholds 1.3·*lastDensity*, 1.6·*lastDensity* and 1.8·*lastDensity*, respectively.

The space used by SEST$_L$ when considering thresholds $1.3 \cdot lastDensity$, $1.6 \cdot lastDensity$ and $1.8 \cdot lastDensity$ were 33Mb, 29Mb and 28Mb, respectively, against 38 Mb required by MVR-tree and 24 Mb required by plain SEST$_L$. Figure 17 indicates that, when considering a threshold $1.6 \cdot lastDensity$, two global snapshots are created (approximately at time instants 10 and 50), which produces an important improvement on the density, reaching values that are similar to the values of density in scenario (1). Similar results are obtained with a threshold $1.3 \cdot lastDensity$. With a threshold 1.8·*lastDensity* in scenario (5), however, the density continues being larger than the density in scenario (1), negatively affecting the time cost of SEST$_L$.

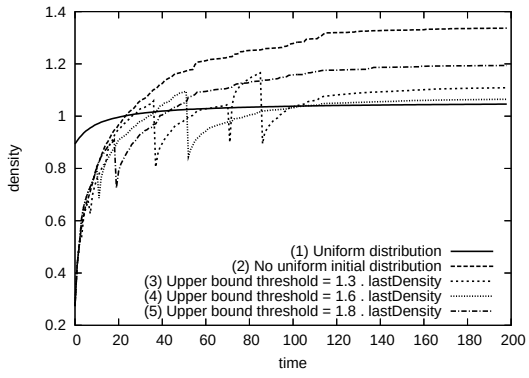


Figure 17: Density for logs of SEST$_L$

Figure 18 shows the query performance of MVR-tree and SEST$_L$ in scenarios (2), (3), (4) and (5). It indicates that, with the thresholds $1.3 \cdot lastDensity$ or $1.6 \cdot lastDensity$, SEST$_L$ outperforms MVR-tree when the length of the query's time interval exceeds 15 time units. The results of these two scenarios are similar to those of in scenario (1) (Figure 4), with an additional storage cost that is still less than the space required by MVR-tree.

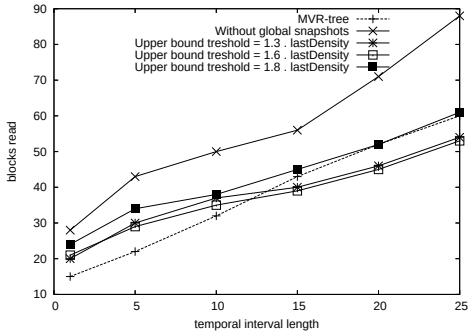


Figure 18: Query performance of SEST$_L$ with global snapshots

7 Conclusion and future work

This work proposes a new spatio-temporal access method, SEST$_L$, that handles events and snapshot associated with space partitions. Based on the experimental results, SEST$_L$ requires between 62% and 85% of the space used by MVR-tree. SEST$_L$ also outperforms MVR-tree for processing queries with time interval over 15 time units(this is an absolute number, which may be rather small or not depending on the application). Unlike other access methods, SEST$_L$ can also answer event-oriented queries efficiently. In addition, SEST$_L$ could be used for other types of queries, such as queries that specify a spatio-temporal pattern as a sequence of distinct spatial predicates in temporal order [8], called spatio-temporal pattern queries (STP). SEST$_L$ can efficiently process STP queries because a *log* in the structure keeps the information about the moment in which an object enters and leaves its assigned space partition.

This paper has also presented and validated a cost model that allows us to evaluate the method without running experiments. The cost model has a good prediction property, having a relative average error of 15% and 8% for storage and time costs, respectively.

As future work, we are developing a method to obtain the values of the parameters of SEST$_L$ such that the structure is optimized with respect to pre-defined constraints of storage or time cost. We will also in-

corporate global snapshots into the cost model and study new algorithms for join and closest-neighboring queries. Finally, we will evaluate the performance of SEST_L for processing STP queries.

References

[1] Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. An asymptotically optimal multiversion b-tree. *The VLDB Journal 5*, 4 (1996), 264–275.

[2] Cole, S., and Hornsby, K. Modeling noteworthy events in a geospatial domain. In *First International Conference, GeoS 2005* (2005), LNCS 3799, Springer, pp. 77–89.

[3] Faloutsos, C., Sellis, T., and Roussopoulos, N. Analysis of object oriented spatial access methods. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1987), ACM Press, pp. 426–439.

[4] Galton, A. Fields and objects in space, time, and space-time. *Spatial Cognition and Computation 4*, 1 (2004), 39–68.

[5] Galton, A., and Worboys, M. Processes and events in dynamic geo-networks. In *First International Conference, GeoS 2005* (2005), LNCS 3799, Springer, pp. 45–59.

[6] Gutiérrez, G., Navarro, G., Rodríguez, A., González, A., and Orellana, J. A spatio-temporal access method based on snapshots and events. In *Proceedings of the 13th ACM International Symposium on Advances in Geographic Information Systems (GIS'05)* (2005), ACM Press.

[7] Guttman, A. R-trees: A dynamic index structure for spatial searchinng. In *ACM SIGMOD Conference on Management of Data* (1984), ACM, pp. 47–57.

[8] Hadjieleftheriou, M., Kollios, G., Bakalov, P., and Tsotras, V. J. Complex spatio-temporal pattern queries. In *VLDB* (2005), pp. 877–888.

[9] Hornsby, K., and Egenhofer, M. Identity-based change: a foundation for spatio-temporal knowledge representation. *International Journal of Geographical Information Science 14*, 3 (2000), 207–224.

[10] Nascimento, M., Silva, J., and Theodoridis, Y. Access structures for moving points. Tech. Rep. TR–33, TIME CENTER, 1998.

[11] Nascimento, M. A., Silva, J. R. O., and Theodoridis, Y. Evaluation of access structures for discretely moving points. In *Spatio-Temporal Database Management* (1999), pp. 171–188.

[12] Tao, Y., and Papadias, D. Efficient historical R-Tree. In *SSDBM International Conference on Scientific and Statical Database Management* (2001), pp. 223–232.

[13] Tao, Y., and Papadias, D. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *The VLDB Journal* (2001), pp. 431–440.

[14] Tao, Y., Papadias, D., and Zhang, J. Cost models for overlapping and multiversion structures. *ACM Trans. Database Syst. 27*, 3 (2002), 299–342.

[15] Theodoridis, Y., and Sellis, T. A model for the prediction of R-tree performance. In *PODS '96: Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1996), ACM Press, pp. 161–171.

[16] Theodoridis, Y., Sellis, T. K., Papadopoulos, A., and Manolopoulos, Y. Specifications for efficient indexing in spatiotemporal databases. In *IEEE Proceedings 10th International Conference on Scientific and Statistical Database Management* (1998), pp. 123–132.

[17] Theodoridis, Y., Silva, J. R. O., and Nascimento, M. A. On the generation of spatiotemporal datasets. In *SSD '99: Proceedings of the 6th International Symposium on Advances in Spatial Databases* (1999), Springer-Verlag, pp. 147–164.

[18] Theodoridis, Y., Stefanakis, E., and Sellis, T. Efficient cost models for spatial queries using R-Trees. *IEEE Transactions on Knowledge and Data Engineering 12*, 1 (2000), 19–32.

[19] Theodoridis, Y., Vazirgiannis, M., and Sellis, T. K. Spatio-temporal indexing for large multimedia applications. In *ICMCS '96: Proceedings of the 1996 International Conference on Multimedia Computing and Systems (ICMCS '96)* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 441–448.

[20] Worboys, M. Event-oriented approaches to geographic phenomena. *International Journal of Geographical Information Science 19*, 1 (2005), 1–28.

[21] Xu, X., Han, J., and Lu, W. RT-tree: An improved R-tree index structure for spatio-temporal database. In *4th International Symposium on Spatial Data Handling* (1990), pp. 1040–1049.