LES-tree: A Spatio-temporal Access Method based on Snapshots and Events [*]

Gilberto A. Gutiérrez
Center for Web Research
University of Chile and
University of Bío-Bío
Avenida La Castilla S/N
Chillán / Chile
ggutierr@ubiobio.cl

Gonzalo Navarro
Center for Web Research and
Department of Computer Science
University of Chile.
Avenida Blanco encalada 2120
Santiago / Chile
gnavarro@dcc.uchile.cl

M. Andrea Rodríguez
Department of Computer Science
University of Concepción
Edmundo Larenas
Concepción / Chile
andrea@udec.cl

October 14, 2008

Abstract

This work presents a new access method (LES-tree) for spatio-temporal databases that handles discrete change events over objects' spatial attributes. The main characteristic of this structure is that, in addition to the traditional database snapshots, LES-tree explicitly stores the events in *log* structures associated with space partitions. The definition of this new access method aims to extend capabilities of current spatio-temporal access methods to queries on *events*, while competing with current structures for traditional *time-slice* and *time-interval* queries. The paper describes the structure and presents favorable experimental cost analyses of the structure.

[*]This work was partially funded by Millennium Nucleus Center for Web Research, grant P04-067-F, Mideplan, Chile. Gilberto Gutiérrez was also funded by research grant 073218 4/R, University of Bío-Bío, Chile

1 Introduction

Spatio-temporal databases are composed of spatial objects that change their location or shape at different time instants [18]. 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. Spatio-temporal applications have been classified into three categories depending on the type of data they manage [12]:

a) Applications that deal with continuous changes, such as the movement of a car on a highway.

b) Applications that involve objects that change their location in space by modifying their shape or by a movement in a discrete manner. An example is the change in the administrative boundary of a city over time.

c) Applications that integrate both previous behaviors. This type of applications appears in the environmental area where it is necessary to model objects' movement and objects' geometric changes over time.

Extending *window/range queries* in spatial databases, the most studied types of queries in spatio-temporal databases are *time-slice* and *time-interval* queries [16]. *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. These queries focus on the coordinate- or snapshot-based representations of objects' movement. In addition to this type of representation, recent studies have emphasized the relevance of handling events, encouraging research in the integration of coordinate- and event-based representation [23, 3, 2]. Event representation enables to manage relationships between events, querying about objects' states, and querying when and why changes on objects occur.

There exist various spatio-temporal access methods that are adequate for applications that handle discrete changes of spatial objects. Some are RT-tree [24], HR-tree (Historical R-tree) [10, 11], 3D R-tree [21], HR$^+$-tree [14], MV3R-tree [15] and OLQ (Overlapping Linear Quadtree) [22], among others. These structures are designed to answer *time-slice* and *time-interval* queries about the history of the spatial attributes of objects. In addition, several of these existing spatio-temporal access methods also handle spatial changes of objects [10, 15, 21], but they use data about these changes with the purpose of updating the underlying data structure. They do not keep data about changes as records of events occurred over objects and, therefore, they cannot efficiently answer queries about events occurred in a time interval.

This work aims to define a new access method that can efficiently answer *time-slice* queries, *time-interval* queries, and *queries on events.* To the best of our knowledge, only the preliminary work in [5] indexes *events* to process traditional time-slice and time-interval queries. In this work, we also address event-based queries that retrieve objects satisfying an event predicate within a spatio-temporal window. For example, retrieve all objects that entered or crossed a given spatial window within a time interval. These queries are easily extended to spatio-temporal pattern queries [7], where we may want to retrieve objects that follow certain patterns of events in a particular sequence.

Our new access method, LES-tree, is based on producing snapshots after a certain number of changes occurred over objects, and on storing the events that produce these changes in a data structure called a *log.* Consequently, LES-tree enables the representation of (1) *temporal snapshots* and (2) *events on objects*, as advocated in [23]. This approach has been briefly discussed in others studies [8, 9], but it has been discarded a priori by arguing that it is not easy to know how many events determine a new snapshot and that extra time is required for query processing. We show in this paper that this is not a serious drawback. The number of snapshots represents a trade-off between space and answer time, since a larger number of snapshots decreases the answer time of a query while increasing the storage space. Inversely, a smaller number of snapshots decreases the space while increasing the answer time; and then, the frequency of snapshots can be adjusted depending on the type of applications and the change frequency of objects. For example, there may be applications where it is not of interest to query about objects' states over some period of time. Our data structures for snapshots and changes are independent, and so are the improvements that can be obtained in either structure. Furthermore, integration of existing spatial access methods for handling snapshots into this approach can be easily achieved.

The idea of snapshots and *logs* has also been used with the purpose of maintaining materialized views in a database [4]. In this context, a *log* is created over the master table, from which the initial view is created. Then, the refreshed view is the result of applying the changes stored in the *log.* These *logs* are eliminated after the actualization of the view, since only the last state of the database is requested. The *logs* in our proposal, in contrast, are a part of the indexing structure, remain over time, and can derive different temporal states of the database.

A preliminary proposal of an access method with these characteristics is the SEST-Index [5]. The idea of SEST-Index consists in maintaining the snapshots of the database for certain time instants (by using an R-tree) and having a global *log* to store the events occurred between 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. Although this structure presents some good properties for time-slice and event queries, it has serious storage cost and scalability problems, and its performance decreases drastically as the number of events increases.

This paper extends and complements substantially the proposal described in [5]. Instead of taking snapshots at the (global) database granularity, it considers snapshots at the region granularity (leaf snapshots). These regions correspond to the space partitions derived from the R-tree structure associated with a global snapshot. The main contributions of this work are:

i) It presents a new spatio-temporal access method based on snapshots and events. This new access method considers snapshots with region granularity. These regions are modified by the use of global snapshots along time.

ii) It presents algorithms not only for time-slice, time-interval, but also for event queries.

iii) It experimentally compares the proposed data structure against MVR-tree [15, 16], MV3R-tree [15], and the preliminary proposal SEST-Index published in [5] for the different times of queries. To the best of our knowledge, MVR-tree and its improved varient MV3R-tree are structures that outperform previous spatio-temporal access methods in terms of time and space requirements for *time-slice* and *time-interval* queries. Results indicate that our spatio-temporal access method has better performance than SEST-Index and competes closely, or even overcomes, MVR-tree, including its improved variant .

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 describes and evaluates a cost model for LES-tree. Section 5 gives experimental evaluations with respect to SEST-Index and MVR-tree. Conclusions and future work are given in Section 6.

2 Spatio-temporal access methods

This section describes the main spatio-temporal access methods available for applications of category (b) (see Section 1) that have been designed to answer *time-slice* and *time-interval* queries. We focus on the MVR-tree/MV3R-tree and SEST-Index structures, against which we compare the new proposed structure. A classification of the existing spatio-temporal access methods is the following: (a) Methods that treat time as another dimension. (b) Methods that incorporate the temporal information in the nodes of the structure without considering time as another dimension. (c) Methods based on overlapping structures. (d) Methods based on multiversioning of the structure. (e) Methods based on snapshots and events.

The 3D R-tree [21] considers time as another axis along with the spatial coordinates. In a three-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 the time interval $[t_i, t_j)$, and then it locates at (x_j, y_j) during the 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 [16].

RT-tree [24] is a structure where the temporal information is kept in the nodes of the R-tree. This is an extension to the data content of a traditional R-tree. In this type of structure, the temporal information plays a secondary role because the search is guided by the spatial information. Thus, queries with temporal conditions cannot be efficiently processed [10].

HR-tree [11, 10] and MR-tree [24] 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 both trees. The major advantage of the HR-tree is its efficiency in processing *time-slice* queries. Its major disadvantage is the excessive space that it requires to store the structure. For example, if only one object of each leaf node moves at instant t_i, the tree is completely duplicated at instant t_{i+1}.

MVR-tree [15, 14, 16] is a structure based on handling multiple versions. It is an extension of MVB-tree [1], where the time-varying attribute is spatial. 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 a MBR. An entry is alive at time instant t if $t_s \leq t < t_e$ and *dead* otherwise. MVR-tree imposes constraints on the number of entries stored in its nodes. A constraint ensures that there exist either zero or at least $b \cdot p_{version}$ alive 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 time instants for processing *time-slice* queries. Other constraints (namely, strong version overflow and strong version underflow) ensure a good space usage in the algorithms for insertion and deletion [15, 14, 16]. Like the MVB-tree, an 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 time when processing short *time-interval* queries. A modification of MVR-tree called MV3R-tree [15] improves the performance of MVR-tree for long *time-interval* queries by adding an auxiliary 3D R-tree for processing these queries. With the purpose of maintaining the storage within reasonable limits, both indices must share the same leaf pages, which makes the insertion algorithm rather complex.

A disadvantage of the MVR-tree is the insertion of artificial entries at the leaf nodes as it does not guarantee that the real lifespan of an object is stored in only one node. For example, if an object O_1 was at position S_1 in a time interval $[1, 20)$, the insertion algorithm of MVR-tree may create two entries $\langle S_1, 1, 8 \rangle$ and $\langle S_1, 8, 20 \rangle$ in two different nodes, making it more difficult to obtain the exact instant when the object O_1 arrives or leaves the position S_1.

SEST-Index [5] is a structure that maintains snapshots for some time instants and stores 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 [5], but it has two important limitations: (i) the objects must be points and (ii) the region where the changes occur must be fixed. The proposal in this paper follows some of the ideas of SEST-Index, but it overcomes the two previous limitations and achieves good space usage, without compromising time efficiency.

3 LES-tree: A Spatio-temporal Access Method Based on Snapshots and Events

Similar to SEST-Index [5], LES-tree maintains snapshots for some time instants and stores 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 [5], but it has two important limitations: (i) the objects must be points and (ii) the region where the changes occur must be fixed. LES-tree overcomes these two limitations and achieves good space usage, without compromising time efficiency.

LES-tree considers two types of snapshots, which handle different space granularity. The first type of snapshots (global snapshot) corresponds to an R-tree (spatial indexing structure) including all objects existing at a particular time instant. The second type of snapshots (leaf snapshot) forms part of the *logs* assigned to leaves of the global snapshots. These *logs* store a sequence of events and several leaf snapshots along time. When the number of events stored in a *log* after a last leaf snapshot exceeds a threshold, a new leaf snapshot is created and stored in the *log*.

Figure 1 shows the general schema of the LES-tree with the two types of snapshots. The objective of global snapshots is to maintain the performance of the query processing along time, since the insertions of events that produce the growing areas of leaves provoke deterioration in the selectivity of the leaf snapshots.

We refer as LES-tree$_l$ to the structure composed of one global snapshot and its corresponding *logs*. Consequently, a LES-tree corresponds to a sequence of LES-tree$_l$ generated from consecutive non-overlapping time intervals of different lengths. These lengths of time intervals are determined automatically by LES-tree and can be adjusted to improve the performance of the indexing structure. In this section we will describe in detail the data structure, the dynamic of the global snapshots, and the update and search algorithms.

3.1 LES-tree structure

The structure of LES-tree considers an array S (see Figure 1) with an entry of type $\langle t_s, pLES\text{-}tree_l \rangle$ for each global snapshot, where t_s corresponds to the time instant in which the R-tree was created and $pLES\text{-}tree_l$ is the reference to the corresponding LES-tree$_l$.

LES-tree$_l$ (see Figure 2), consists of an R-tree [6] and a set of *logs* assigned to the regions of leaves in the R-tree. Here a *log* (leaf) is a structure that stores leaf snapshots and events. In this structure, the movement of an object is not inserted directly in the R-tree, producing the classical node splitting of the R-tree, but they are inserted as events in a log of an R-tree's leaf. Figure 2 presents a region A and its corresponding *log* with three objects at instant t_0. At instant t_1, region A has grown to include a fourth object, what is reflected on the corresponding leaf snapshot. The changes that occurred between t_0 and t_1 are stored as events in the *log* associated with A.

In LES-tree$_l$, 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 areas of non-leaf nodes in the R-tree increase and the efficiency of query processing degrades (see Figure 2), a problem that Section 3.2 addresses.

The LES-tree$_l$ considers an R-tree [6], where the leaves are *logs* and these *logs* are linked lists of blocks. A *log* has two types of entries: event or change entries and leaf snapshot entries (the first entry is always a leaf snapshot). The 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 on the geometry 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 as a *move_out*, and an object movement as a *move_out* followed by a *move_in*. The structure that stores *move_out* entries could only include attributes t and Oid. This decreases the storage cost of the structure, at the price of increasing the time cost for processing queries on events (but not the classical *time-slice* and *time-interval* queries). Later, in Section 5, we will analyze how much space (and also time) can be saved if the structure only supports *time-slice* and *time-interval* queries.

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

Like SEST-Index [5], each LES-tree$_l$ also uses a parameter d, equal for all *logs* in the structure,

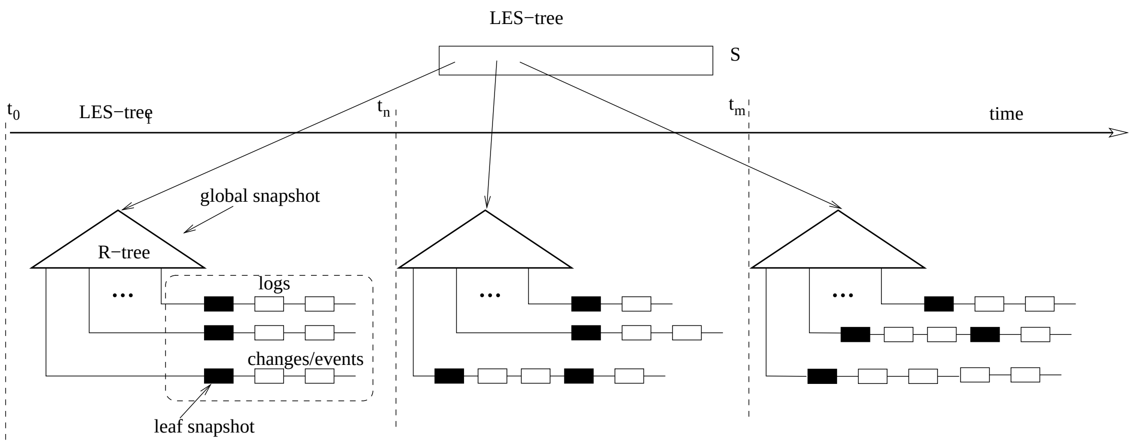


Figure 1: General outline of LES-tree

which represents the amount of memory, measured in the number of blocks, used to store events between consecutive *leaf* snapshots.

3.2 Dynamic regions

A potential problem of LES-tree$_l$ is that the query selectivity can deteriorate when regions grow due to the arrival of new events that are stored in their *logs*. Remind that inserting an entry of type $move\_in$ can expand the area of MBRs, from the leaves (*logs*) to the root of the tree, whereas a $move\_out$ does not reduce the area of their corresponding MBR. As a consequence of this ever growing area of MBRs, the time cost of a spatio-temporal query (Q, t) submitted at instant $t_1 \geq t_s$ is lower than the time cost of the same query submitted at instant t_2, with $t_2 > t_1$.

To measure this effect, we define a *density* measure associated with the current R-tree. This measure $realDen$ is defined by Eq. (1), where M is the set of MBRs located at the leaves of the R-tree, and $TotalArea$ is the workspace area. $realDen$ describes how compactly the leaf MBRs cover the space, so a lower $realDen$ value implies that the database points are covered by smaller MBRs. The value of $realDen$ affects the query performance, since the larger the MBRs are, the larger is the number of *logs* that must be processed to solve a query. The value of $realDen$ usually grows with $move\_in$ events, as MBRs grow and the workspace tends to stay the same. This explains why a query submitted at instant t_1 costs less than a query submitted at instant t_2, with $t_2 > t_1$.

$$realDen = \frac{\sum_{i \in M} Area_i}{TotalArea} \tag{1}$$

To solve the problem of selectivity for dynamic regions, LES-tree uses *global* snapshots which are created when the insertions of new events produces values of $realDen$ (Eq. (1)) larger than a given threshold.

3.3 Operations and algorithms

3.3.1 Updating the structure

The update of LES-tree considers two algorithms. The first one (Algorithm 1) updates the LES-tree$_l$ with the arrival of new events since the time instant in which the global snapshots was created. The second algorithm (Algorithm 2) creates the global snapshots.

Updating LES-tree$_l$. This algorithms updates the structure upon changes that occur at each time instant. Let us assume that the changes to be processed are stored in a list. When an object moves,

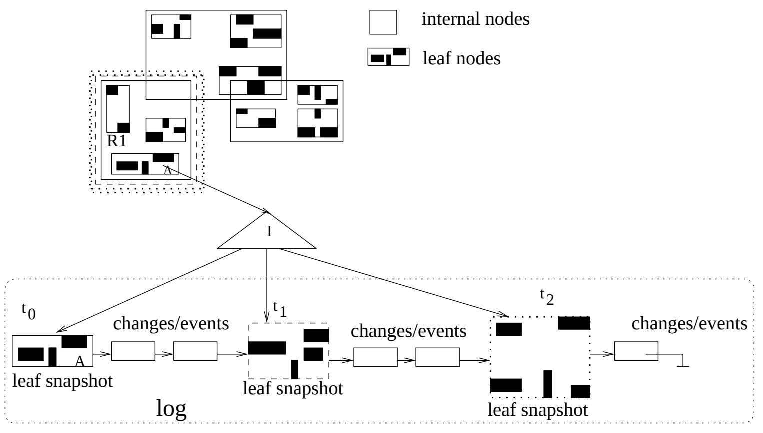


Figure 2: General outline of LES-tree$_l$

two events, *move_out* and *move_in*, are created. The event *move_in* includes attribute values t, *Geometry* and *Oid* of the incoming object. For each *move_in* event, we choose the corresponding *log* where the event should be inserted according to the classical R-tree insertion policy [6] (called *chooseleaf()* in Algorithm 1).

Although the data input for a *move_out* event only contains the attribute values t and *Oid*, the event entry in the log must store the *Geometry* of the object (the geometry of the corresponding move_in event) to enable the efficient process of event-based queries. In addition, this move_out event should be stored in the same log than the corresponding move_in event to reduce the process of time-instant or time-interval queries. As the R-tree shape may have changed (MBRs may have expanded) since the last *move_in* event, *chooseleaf()* may choose a different *log*, even if we have the same geometry for move_in and move_out events. A solution to both problems is to keep a hash table ($\langle Oid, block\rangle$), with *block* being the reference to the current block in the *log* where the last move_in event of the object was stored. The procedure to find the leaf using the hash table is called *choosepreviousleaf()* in Algorithm 1.

Both *move_in* and *move_out* events are inserted as event entries after the last leaf snapshot that was stored in the corresponding *log*. If, just before insertion, the number of changes exceeds parameter d, a new snapshot is created. Note that even though the number of changes may exceed the capacity of the *log* (parameter d), all changes occurred at the same time instant in the region assigned to a *log* are stored as event entries between two consecutive leaf snapshot. The insertion of a *move_in* event may require updating the MBRs of the leaf as well as of the ancestor nodes whose MBRs must now include the *Geometry* of the arriving object.

Creating global snapshots. We propose to use the density value *realDen* to determine the time instant when a new global snapshot should be created. The process of creating a new global snapshot is described in Algorithm 2. Unlike *InsertChanges()*, *UpdateLES-tree()* requires to process together all changes occurred at the same time instant. This guarantees that those changes will not be split between different global snapshots. However, the algorithm can be easily modified to handle sets of changes that span different consecutive time intervals and still enforce the non-splitting property.

The algorithm keeps two R-trees. The first one is the last global snapshot that was created (R in the Algorithm 2), and the second one keeps the last position of alive objects in time (*newR-tree* in

```
UpdateLES-tree$_l$(Changes $C$, LES-tree$_l$ $L$, Integer $d$) {$C$ is the list of changes with non decreasing times and $d$ is the capacity of a log to store event entries between leaf snapshots}
Let $R$ the R-tree of LES-tree$_l$ $L$
for each $c \in C$ do
    if $c.Op = move\_in$ then
        $b = chooseleaf(R, c.Geometry)$
    else
        $b = choosepreviousleaf(R, c.Oid, c.Geometry)$.
    end if
    Let $L$ be the list of events occurred in $b$ after the last leaf snapshot.
    Let $l$ be the number of changes stored in $L$.
    if $l > d \ \wedge c.t \neq L[l].t$ then
        Create a new leaf snapshot $S$ in $b$
        Create a new empty list $L$ and assign it to $S$
    end if
    Insert $c$ at the end of $L$
    if $c.Op = move\_in$ then
        Update the MBRs of all R-tree nodes in the path followed by chooseleaf() to reach $b$.
    end if
end for
```

Algorithm 1: Algorithm to update the structure

algorithm 2). The arriving events update both R-trees, but in different way. While $InsertChange()$ updates the first R-tree (the global snapshot) with the arrival of new events, the classical insertion and elimination algorithms of the original R-tree updates the second R-tree. The cost of storing this second R-tree tends to be constant and marginal with respect to the total storage cost of the structure. This is particular true in cases with a large number of objects that move along several time instants, and for which the storage of several global snapshots and *logs* largely exceeds the space used by a single R-tree.

Algorithm 2 checks that the value of *realDen* after inserting new change events does not exceed the threshold $ls \cdot lastDensity$, where *lastDensity* is the density of the last global snapshot and $ls \geq 1$ is a parameter of the structure. When the value of *realDen* exceeds the threshold, a new R-tree is created. Let *newDensity* be the density of the R-tree just created. The algorithm for updating LES-tree ensures $newDensity < li \cdot realDen$ before actually creating the new R-tree; otherwise, the improvement is not worth the extra space. Here, $li \leq 1$ is another parameter of the algorithm.

```
UpdateLES-tree( array $S$, float $lastDensity$, Changes $C$, float $ls$, float $li$, Integer $d$, R-tree $newR\text{-}tree$ )
{$S$ is the array of entries to existing global snapshots, $lastDensity$ corresponds to the value of $realDen$ of the last R-tree in $S$, $C$is a list with changes that occurred at the last time instant, $ls$ and $li$ are fractions of $lastDensity$ }
Let $E$ be the last entry of array $S$ and $L = E.LES\text{-}tree_l$
UpdateLES-tree$_l$(C, L, d) {Insert changes in the $L$ using Algorithm 1}
$UpdatenewR\text{-}tree(C, newR\text{-}tree)$ {Updates $newR\text{-}tree$ with changes in $C$ list}
Let $newDensity$ be the new value of $realDen$ for $L.R\text{-}tree$ after inserting the changes
if $newDensity > ls \cdot lastDensity$ then
    Let $tmpDensity$ the value of $realDen$ for $newR\text{-}tree$
    if $tmpDensity < li \cdot newDensity$ then
        Create a new LES-tree$_l$ $NL$ from the $newR\text{-}tree$
        $E1 = \langle t, NL \rangle$
        Add entry $E1$ to $S$
        $newR\text{-}tree$ = Duplicate($newR\text{-}tree$)
    end if
end if
```

Algorithm 2: Algorithm that controls the creation of global snapshots

3.3.2 *Time-slice* queries

To process a *time-slice* query (Q, t), the first step is to find the entry i in the array S such that S_i.LES-tree$_l$ includes time t and is the last entry such that $S_i.t_s \leq t$. Then, the algorithm recovers all leaves of the global snapshot associated with S_i.LES-tree$_l$ that intersect with Q. Next, for the *log* of each leaf, the process obtains the corresponding leaf 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 leaf snapshot that 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 3). The whole answer is the union of the results obtained from each involved leaf.

```
1:  time-sliceQuery(Rectangle $Q$, Time $t$, array $S$)
2:  Find the last entry $i$ in $S$ such that $S_i.t_s \leq t$
3:  Let $R$ be the R-tree of $S_i$.LES-tree$_l$.
4:  $B = SearchRtree(Q, R)$ {$B$ is the set of leaves ($logs$) that intersect $Q$}
5:  $G = \emptyset$ {$G$ is the set of objects that belong to the answer}
6:  for each $log$ $b \in B$ do
7:     Let $tr$ be the time of the latest leaf snapshot in $b$ such that $tr \leq t$
8:     Let $A$ be the set of all objects in the leaf snapshot created at time instant $tr$ in $log$ $b$ and that intersect $Q$
9:     for each event entry $c \in b$ such that $tr < c.t \leq t$ do
10:       if $c.Geometry$ $intersects$ $Q$ then
11:          if $c.Op = move\_in$ then
12:             $A = A \cup \{c.Oid\}$
13:          else
14:             $A = A - \{c.Oid\}$
15:          end if
16:       end if
17:    end for
18:    $G = G \cup A$
19: end for
20: return $G$
```

Algorithm 3: Algorithm to process a *time-slice* query

3.3.3 *Time-Interval* queries

Two procedures process *time-interval* queries $(Q, [t_i, t_f])$. The first one (see Algorithm 4) aims to transform $(Q, [t_i, t_f])$ into a set of *time-interval* sub-queries (Q, t_1, t_2), whose time intervals $[t_1, t_2]$ are non-overlapping sub-intervals of $[t_i, t_f]$. The limits of an interval are defined such that the sub-query covers the time interval between consecutive global snapshots, with the exception that the initial instant of the first time interval must be t_i, and the final instant of the last time interval must be t_f. The answer of the query $(Q, [t_i, t_f])$ is obtained by the algorithm 5 as the union of the answers of each of the sub-queries.

The second procedure (see Algorithm 5) processes each of the *time-interval* $(Q, [t_1, t_2])$. This algorithm starts by finding the set of spatial objects that intersect the query window (Q) at the initial instant t_1. This is equivalent to a *time-slice* query at instant t_1. Then, objects are updated based on the changes occurred within the interval $(t_1, t_2]$ (Algorithm 5).

```
1:  IntervalQuery(Rectangle $Q$, Time $t_i, t_f$, array $S$)
2:  $ANS = \emptyset$ {Answer set}
3:  Find the last entry $i$ in $S$ such that $S_i.t_s \leq t_i$
4:  $t_1 = S_i.t_s$
5:  while $t_1 < t_f$ do
6:     if $i$ is the last entry in $S$ then
7:        $t_2 = t_f$
8:     else
9:        $t_2 = \min(S_{i+1}.t_s, t_f)$
10:    end if
11:    $ANS = ANS \cup$ SubIntervalQuery($Q$, Time $t_1, t_2$, $S_i$.pLES-tree$_l$)
12:    $t_1 = t_2$
13:    $i = i + 1$
14: end while
15: return $ANS$
```

Algorithm 4: Algorithm to process a *time-interval* query

3.3.4 Event queries

One of the novelties of the LES-tree structure is its capability for processing not only *time-slice* and *time-interval*, but also queries on *events*. 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, for example, in applications that aim to analyze the pattern of objects' movements [23, 3].

Processing event queries with LES-tree (see Algorithm 6) is simple and efficient, since the structure explicitly stores the changes over objects' geometry. Algorithms for these types of queries are similar to those for *time-slice* and *time-interval* queries.

4 A LES-tree Cost Model

This section presents a cost model for LES-tree, which allows us to predict its storage and time costs for spatio-temporal queries. The cost model is experimentally validated to demonstrate its prediction capability.

The cost model of LES-tree assumes that the initial locations of objects and their subsequent positions upon movements distribute uniformly and that the events consist of random changes in objects' location,

```
1: **SubIntervalQuery(Rectangle $Q$, Time $t_1$, $t_2$, LES-tree$_l$ $L$ )**
2: Let $R$ be the R-tree of $L$
3: $B = SearchRtree(Q, R)$ {$B$ is the set of leaves (*logs*) that intersect $Q$}
4: $G = \emptyset$ {$G$ is the set of objects that belong to the answer}
5: **for** each *log* $b \in B$ **do**
6:   Let *tr* be the time of the latest leaf snapshot in $b$ such that $tr \leq t$
7:   Let $A$ be the set of all objects in the leaf snapshot created at time instant *tr* in log $b$ and that intersect $Q$
8:   Update $A$ with the changes stored in $b$ occurred between $(tr, t_1]$ {like a *time-slice* query}
9:   $ts = Next(t_1)$ {$Next(x)$ returns the next instant after $x$ having changes in log $b$}
10:  **while** $t_s \leq t_2 \wedge$ there exist unprocessed event entries in log $b$ **do**
11:    **for** each event entry $c \in b$ such that $c.t = t_s$ **do**
12:      **if** *c.Geometry intersects Q* **then**
13:        **if** $c.Op = move\_in$ **then**
14:          $A = A \cup \{c.Oid\}$
15:        **else**
16:          $A = A - \{c.Oid\}$
17:        **end if**
18:      **end if**
19:    **end for**
20:    let $G = G \cup \{\langle o, ts\rangle,\ o \in A\}$
21:    $ts = Next(ts)$
22:  **end while**
23: **end for**
24: **return** $G$
```

Algorithm 5: Algorithm to process a subquery.

so the number of objects does not change along time. Figure 3 describes the variables used in the cost model of LES-tree.

The development of the model follows two steps. First, a model for LES-tree$_l$ and then its extension to a model for LES-tree.

4.1 A cost model for LES-tree$_l$

4.1.1 Storage cost of the R-tree

Let N be the number of objects stored in an R-tree with average fanout f. The height h of an R-tree is given by the equation $h = \lceil \log_f N \rceil$ [17].

Since the number of entries in a node is on average 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

```
1: **EventQuery(Rectangle $Q$, Time $t$, array $S$)**
2: Find the last entry $i$ in $S$ such that $S_i.t_s \leq t$
3: Let $R$ be the R-tree of $S_i$.LES-tree$_l$.
4: $B = SearchRtree(Q, R)$ {$B$ is the set of regions that intersect $Q$}
5: $oi = 0$ {number of objects that moved into $Q$ at instant $t$}
6: $oo = 0$ {number of objects that moved out of $Q$ at instant $t$}
7: **for** each *log* $b \in B$ **do**
8:   find the first event entry $c \in b$ such that $c.t = t$
9:   **while** $c.t = t$ **do**
10:    **if** *c.Geometry intersects Q* **then**
11:      **if** $c.Op = move\_in$ **then**
12:        $oi = oi + 1$
13:      **else**
14:        $oo = oo + 1$
15:      **end if**
16:    **end if**
17:    $c = NextChange(c)$ {$NextChange(x)$ returns the event entry following $x$ in log $b$}
18:  **end while**
19: **end for**
20: **return** $(oi, oo)$
```

Algorithm 6: Algorithm to process an event query

nodes at the level immediately superior to the leaves is $N_2 = \left\lceil \frac{N_1}{f} \right\rceil$. Considering that the root is at level h and the leaves are at level 1, the average number of nodes at level j is given by equation $N_j = \left\lceil \frac{N}{f^j} \right\rceil$ [17]. Thus the average number of nodes used (i.e., storage cost) for an R-tree is determined by Eq. (2).

$$\begin{aligned} TN &= \sum_{j=1}^{h} \left\lceil \frac{N}{f^j} \right\rceil \approx h + \left\lfloor N \cdot \frac{(f^h - 1)}{f^h \cdot (f-1)} \right\rfloor \\ &\approx \log_f N + \frac{N}{f} \end{aligned} \tag{2}$$

4.1.2 Time cost of the R-tree

Based on [20, 17], the number of nodes accessed by an R-tree (i.e., time cost) in spatial window queries $(q_1, \ldots, q_n)$ is given by Eq. (3) (see next for D_j).

$$DA_n = 1 + \sum_{j=1}^{h} \frac{N}{f^j} \cdot \prod_{i=1}^{n} \left(\left(D_j \cdot \frac{f^j}{N} \right)^{\frac{1}{n}} + q_i \right) \tag{3}$$

Symbol	Description
ai	Length of the time interval of a query (subquery)
c	Changes that fit in a block (one change is equivalent to two operations, one *move_in* plus one *move_out*)
cc	Usage average percentage of a node in an R-tree
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), $f = cc \cdot M$
h	Height of an R-tree
il	Number of time instants that can be stored in a *log* between consecutive leaf snapshots
l	Total number of changes stored between consecutive leaf snapshots
igs	Number of time instants between global snapshots, or how long it takes to create a new global snapshot
M	Maximum capacity of a node in an R-tree (maximum number of entries)
M_1	Maximum capacity of a node in an R-tree for reaching a given density
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	Width of the query rectangle along a dimension, as a fraction of the total space
TN	Total number of blocks used by an R-tree
ls	Threshold that determines when a new global snapshot is created
NB	Number of *logs*
DA_{ts}	Number of blocks accessed in a *time-slice* query
DA_{in}	Number of blocks accessed in a *time-interval* query
TB_{total}	Total number of blocks needed to store the LES-tree$_l$ structure

Figure 3: Definition of variables

Given that our experiments consider 2-dimensional objects, $DA = DA_2$ is defined by Eq. (4), assuming a square query window, $q = q_1 = q_2$.

$$DA = 1 + \sum_{j=1}^{h} \left(\sqrt{D_j} + q \cdot \sqrt{\frac{N}{f^j}} \right)^2 \tag{4}$$

In Eqs. (3) and (4), D_j corresponds to the *density* of spatial objects at level j [20, 17]. This is obtained by Eq. (5), with D_0 being the density of the spatial objects that are indexed in the R-tree.

$$D_j = \left(1 + \frac{\sqrt{D_{j-1}} - 1}{\sqrt{f}} \right)^2, 1 \leq j \leq h \tag{5}$$

An important property of Eqs.(3) and (4) 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. In this paper we have used $D_0 = 0$, as we index points. In general, D_0 can be computed by using Eq. (1) (i.e., $D_0 = realDen$), where M represents the objects to be indexed.

4.1.3 Storage cost of LES-tree$_l$

Two values describe the storage cost (number of blocks) used by the *logs* of LES-tree$_l$: the number of *logs* and the storage needed per *log*. The number of *logs* is equal to the number of leaves in the R-tree, which is expressed by equation $NB = N_1 = \left\lceil \frac{N}{f} \right\rceil$.

The number of time instants (il) that can be stored between leaf snapshots is determined by Eq. (6). In this equation, l is the capacity of the *log* to store change entries between two consecutive leaf snapshots, and $p \cdot f$ represents the average number of changes that would occur in a region at each time instant.

$$il = \left\lceil \frac{l}{p \cdot f} \right\rceil \tag{6}$$

Using Eq. (6), it is possible to calculate the value *nl* (Eq. (7)), such that all events occurred at the same time instant are stored between the same leaf snapshots.

$$nl = p \cdot f \cdot il \tag{7}$$

With *il* and *nl*, the number of blocks used per each *log* is given by Eq. (8). In this equation, the first term of the sum corresponds to the number of leaf snapshots. This is equal to the number of blocks used by a leaf snapshot if we assume that the alive objects fit, on average, in a disk block (as it initially happens in the R-tree leaves). The second term represents the number of blocks required, on average, for each *log* to store all the changes occurred at all time instants except those after the last leaf snapshot. The last term is the number of blocks occupied for storing the changes occurred after the last leaf snapshot.

$$\begin{aligned} TB_{log} &= \left\lceil \frac{nt}{il} \right\rceil + \left(\left\lceil \frac{nt}{il} \right\rceil - 1 \right) \cdot \left\lceil \frac{nl}{c} \right\rceil \\ &+ \left\lceil \left(\frac{nt}{il} - \left\lfloor \frac{nt}{il} \right\rfloor \right) \cdot \frac{nl}{c} \right\rceil \end{aligned} \tag{8}$$

Finally, the number of blocks used by the LES-tree$_l$ is given by Eq. (9).

$$\begin{aligned} TB_{total} &= (TN - NB) + NB \cdot TB_{log} \\ &\approx \log_f N + N \cdot nt \cdot p \left(\frac{1}{l} + \frac{1}{c} \right) \end{aligned} \tag{9}$$

The latter approximation is meant to give intuition on the result, and not replace the exact formula. The experimental validation of the model uses the exact formula of TB_{total}, not the approximation.

4.1.4 Time cost of LES-tree$_l$

The time cost of LES-tree$_l$ 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 $h_p = \lceil \log_f N_p \rceil = h - 1$ be the height of the Rp-tree. The number of nodes accessed in the tree is given by Eq. (10).

$$\begin{aligned} DA_{Rp-tree} &= 1 + \sum_{j=1}^{h_p} \left(\sqrt{D_{j+1}} + q \cdot \sqrt{\frac{N_p}{f^j}} \right)^2 \\ &= DA - \left(\sqrt{D_1} + q \cdot \sqrt{\frac{N}{f}} \right)^2 \end{aligned} \tag{10}$$

In Eq. (10), we use D_{j+1} because the objects in the Rp-tree are the leaves of the R-tree. The number of *logs* to be processed in a query depends strongly on the density formed by the MBRs of the leaves and query. This number of *logs* is defined in Eq. (11), where $D_1 = \left(1 + \frac{\sqrt{D_0} - 1}{\sqrt{f}} \right)^2$.

$$NL = \left(\sqrt{D_1} + q \cdot \sqrt{\frac{N}{f}} \right)^2 \tag{11}$$

Therefore, the number of blocks accessed in a *time-slice* query is defined by Eq. (12).

$$\begin{aligned} DA_{ts} &= DA_{Rp-tree} + NL \cdot \left(1 + \left\lceil \frac{nl}{2 \cdot c} \right\rceil \right) \\ &\approx q^2 \cdot \frac{N}{f} \cdot \left(1 + \frac{l}{2 \cdot c} \right) \end{aligned} \tag{12}$$

Likewise, the average number of blocks accessed in a *time-interval* query is given by Eq. (13).

$$\begin{aligned} DA_{in} &= DA_{ts} + NL \cdot \left\lceil \frac{(ai - 1) \cdot p \cdot f}{c} \right\rceil \\ &\approx q^2 \cdot \frac{N}{f} \cdot \left(1 + \frac{\frac{l}{2} + ai \cdot p \cdot f}{c} \right) \end{aligned} \tag{13}$$

Again, the approximations for DA_{ts} and DA_{in} just serve to give intuition and do not replace the exact formulas.

Based on the cost model for *time-slice* and *time-interval* queries, we can derive the cost of processing queries on *events*, since these queries also consider temporal and spatial windows.

4.1.5 Experimental validation of the cost model for LES-tree$_l$

In order to evaluate the cost model, several experimental evaluations were conducted with synthetic data obtained from GSTD [19]. These experiments used 23,268 objects (points) and 200 time instants with change frequencies of 1%, 5%, 10%, 15%, 20% and 25%. They also considered values of the parameter d equal to 2, 4 and 8 blocks, where $l = d \cdot c$ (we consider $c = 21$ changes in our experiments). The value f of the R-tree was set to 34 (68% of the capacity [17] of a node in an R*-tree that is able to hold 50 entries).

Figures 4, 5 and 6 evaluate the prediction capability of the cost model for both storage and time requirements. Figure 4 shows 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%.

Figures 5 and 6 show that the prediction of the time cost of the query processing is very good, with a relative average error of 8%.

4.2 A cost model for LES-tree

This section describes the storage and time costs of LES-tree. The idea of the global snapshots is to keep a stable performance of the structure along time, since the performance decreases as consequence of the increasing density of the initial R-tree. The cost model assumes objects uniformly distributed over the space and the same density of the R-tree in each global snapshot. Another important consideration is that, with the purpose of capturing the evolution of the MBRs' density, the operations of type *move_in* are modeled as insertions in an R-tree. Finally, we assume a constant number of objects along time.

To incorporate the effect of the global snapshots in the basic model for LES-tree$_l$, it is necessary to estimate how many global snapshots (ngs) should be created. To do so, we estimate the number of objects (M_1) that must be stored in each leaf node in order to reach the density threshold $D_1' = ls \cdot D_1$, where D_1 is the initial density of the R-tree and D_1' is the density value that triggers the creation of a new global snapshot.

Given $f = cc \cdot M$ and Eq. (5), Eq. (14) estimates the density of the MBRs of the leaves in the R-tree.

$$D_1 = \left(1 + \frac{\sqrt{D_0} - 1}{\sqrt{cc \cdot M}}\right)^2 \qquad (14)$$

Thus, it is possible to obtain M_1 by Eq. (15).

$$\left(1 + \frac{\sqrt{D_0} - 1}{\sqrt{cc \cdot M_1}}\right)^2 = ls \cdot \left(1 + \frac{\sqrt{D_0} - 1}{\sqrt{cc \cdot M}}\right)^2 \qquad (15)$$

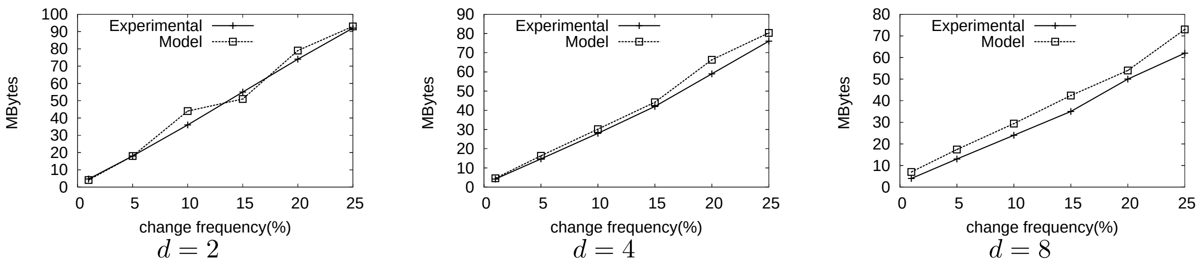
From Eq. (15), M_1 can be expressed by Eq. (16).

$$M_1 = \frac{\left(\sqrt{D_0} - 1\right)^2}{cc \cdot \left(\sqrt{ls} \cdot \left(1 + \frac{\sqrt{D_0}-1}{\sqrt{cc \cdot M}}\right) - 1\right)^2} \qquad (16)$$

As it was mentioned before, the value of M_1 indicates how many events should be inserted to reach density D_1'. With M_1, it is possible to obtain the time interval between global snapshots (igs). To do so, we obtain the total number objects stored in the R-tree when the number of objects in the leaf nodes of the R-tree reaches M_1. This is $M_1 \cdot \frac{N}{cc \cdot M}$, where $\frac{N}{cc \cdot M}$ is the number of leaves (or *logs*) in the original R-tree. Therefore, the number of changes that should be inserted in the R-tree to reach density D_1' is calculated as $M_1 \cdot \frac{N}{cc \cdot M} - N$. The estimated number of objects inserted in the R-tree in every time instant is $N \cdot p$, and therefore, $igs = \left\lceil \frac{M_1 \cdot \frac{N}{cc \cdot M} - N}{N \cdot p} \right\rceil$ is the number of time instants that are stored in a *log* before a new global snapshot is created. This is simplified to obtain Eq. (17).

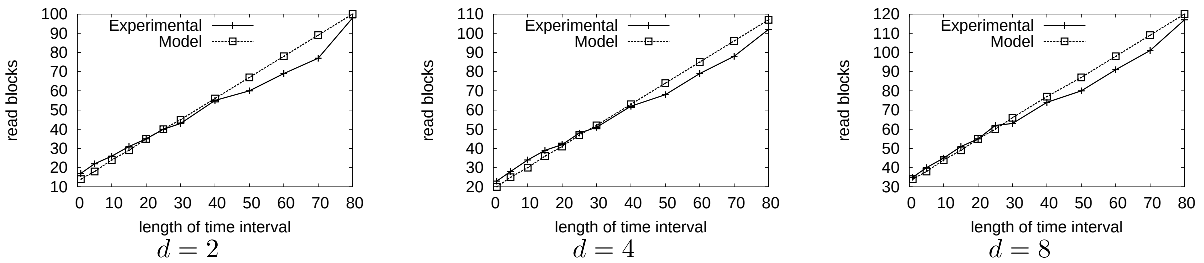
$$igs = \left\lceil \frac{\frac{M_1}{cc \cdot M} - 1}{p} \right\rceil \qquad (17)$$

For example, if $cc = 0.68$, $N = 20,000$, $M = 50$, $ls = 1.3$, $p = 0.10$ and $D_0 = 0$, the values of M_1 and igs are approximately 480 and 132, respectively, which implies that every 132 time instants we need to create a new global snapshot.



$d = 2$ $d = 4$ $d = 8$

Figure 4: Estimation of the storage use for a LES-tree$_l$



$d = 2$ $d = 4$ $d = 8$

Figure 5: Estimation of the time cost of queries by LES-tree$_l$ (10% change frequency and 6% in each dimension for the query window)

4.2.1 Storage cost of LES-tree

Having igs, the estimation of the storage cost for LES-tree is simple, since it is enough to consider that the number of global snapshots is equal to $ngs = \lceil \frac{nt}{igs} \rceil$. Each global snapshot should store the changes occurred along igs time instants. In addition, the last global snapshot should store the changes occurred along $lt = nt - igs \cdot \lfloor \frac{nt}{igs} \rfloor$ time instants. Using Eq. (9), and replacing TB_{log} by TB_{log}^{igs} or TB_{log}^{lt}, we obtain the storage used by the *logs* in each global snapshot. TB_{log}^{igs} is obtained from Eq. (8) by replacing nt by igs, and TB_{log}^{lt} by replacing nt by lt. Thus, the storage cost of LES-tree is defined by Eq. (18).

$$TBGS_{total} = \left\lfloor \frac{nt}{igs} \right\rfloor \cdot \Big((TN - NB) + NB \cdot TB_{log}^{igs}\Big) + (TN - NB) + NB \cdot TB_{log}^{lt} \quad (18)$$

4.2.2 Time cost of LES-tree

Clearly, the cost model defined by Eq. (12) for *time-slice* queries does not change, because to process a query of this type only needs to access a single LES-tree$_l$. For *time-interval* queries, we first need to know the number of global snapshots (i.e., the number of LES-tree$_l$) that the algorithm of a time interval query should access.The estimated number of global snapshots that a query with a time interval of ai intersects is defined by Eq. (19).

$$ngsi = \left\lceil \frac{ai}{igs} \right\rceil \quad (19)$$

The length of the interval ai is transformed into

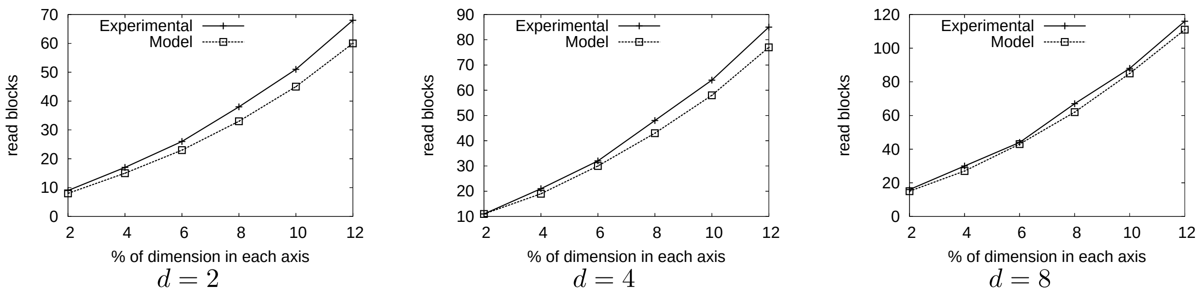


Figure 6: Estimation of the time cost of queries by LES-tree$_l$ (10% change frequency and time interval length equal to 10)

a set of $ni = \left\lfloor \frac{ai}{igs} \right\rfloor$ intervals of length igs plus an additional interval of length $lr = ai - ni \cdot igs$. Thus, to determine the time cost of LES-tree, a *time-interval* query is decomposed into ni queries with time intervals of length igs and a query with time interval of length lr. For each R-tree, the density of leaves' MBR is at worst $D'_1 = ls \cdot D_1$ and, therefore, the number of *logs* that the spatial component of the query will intersect is given by Eq. (20).

$$NL = \left(\sqrt{D'_1} + q \cdot \sqrt{\frac{N}{cc \cdot M}} \right)^2 \qquad (20)$$

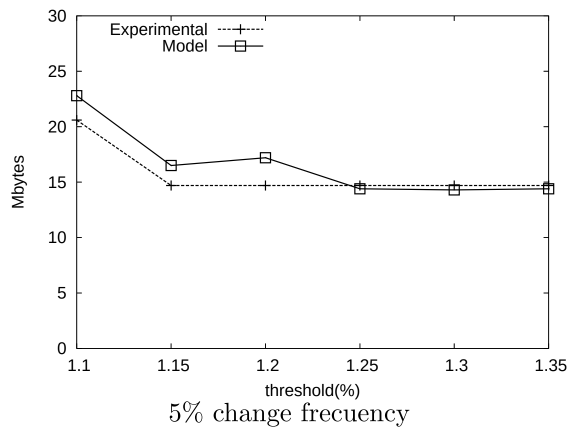
Finally, replacing ai by igs (to obtain DA^{igs}_{in}) or lr (to obtain DA^{lr}_{in}) in Eq. (13), we can obtain the number of blocks accessed by LES-tree in a spatio-temporal query (Eq. (21)).

$$DASG_{in} = ni \cdot DA^{igs}_{in} + DA^{lr}_{in} \qquad (21)$$

4.2.3 Experimental validation of the cost model of LES-tree

In order to evaluate the cost model, new experimental evaluations were conducted with synthetic data obtained from GSTD [19]. These experiments used 23,268 objects (points) and 200 time instants with change frequency of 5% and 10%, and 4 blocks for parameter d. The experiment considered values of the threshold (ls) from 1.1 to 1.35. Figure 7 shows that the prediction of the storage cost is very good, with a relative average error of 14%. The model estimates the time cost of query processing with a relative average error of 14% (see Figures 8 and 9). Note that in Figure 8 the use of global snapshots appears to be counterproductive. This is because we purposely used ls values lower than advisable in order to obtain a clear difference with respect to the standard structure. As the distribution is uniform, a correct parameterization (such as $ls = 1.20$) would recommend very few global snapshots and the difference with the standard structure would be negligible. The use of global snapshots is advantageous when the data does not distribute uniformly, as shown in Section 5.2.

Our model turns also to be useful on nonuniform data. The reason is that the global snapshot technique, in some sense, makes the structure behave over nonuniform data in a way similar to uniform data. To adapt our model to an arbitrary initial distribution, we only have to compute the initial density and global snapshot rate according to the real data. For the density, instead of using Eq. 14, we must build the R-tree for the spatial data at hand, and use the MBRs M of the leaves to compute D_1, as *realDen* of Eq. (1). For the global snapshot rate we need to compute M_1 for arbitrary distributions. We replace the analytic formula of Eq. (16) by a procedure that builds R-trees over the initial data using increasing node capacities until



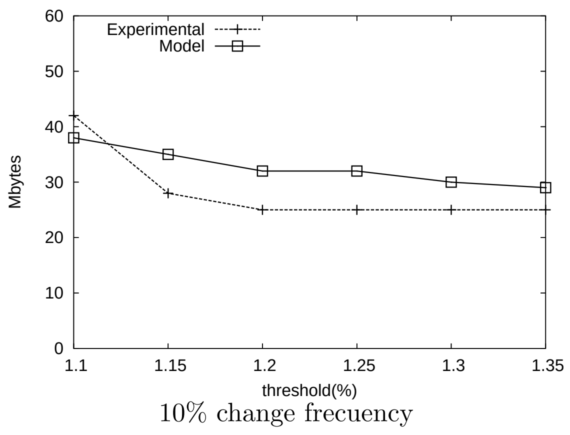


Figure 7: Estimation of the storage used for LES-tree

the density reaches a value $D'_1 = ls \cdot D_1$. This is all the computation we need over the real data (no simulations over query streams nor change events, for example). All the rest of the formulas of the model can be used unaltered once D_1 and M_1 are obtained.

An experimental validation of our model in a scenario with non uniform distribution of objects and global snapshots uses the same data set of the evaluation of MVR-tree in [16]. Such data set consists of 23,268 objects (points) that moves with 10% of change frequency during 199 time instants. We refer to this data set as NUD (Figure 10).

In Figure 11 we compare the experimental behavior over the NUD data with our model. It can be seen that our model, although designed for uniform distributions, accurately predicts the behavior of the structure on nonuniform data as well. The space required is accurately predicted as well: the error is typically very low (e.g. 3% for $ls = 1.3$). For very low ls the space required can get larger (e.g. 20% for $ls = 1.10$), but those low ls values are not of use in practice.

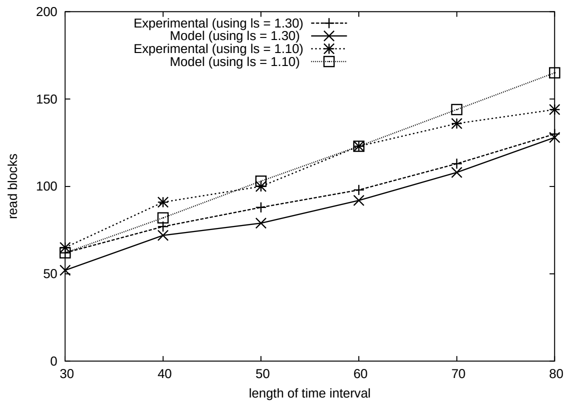
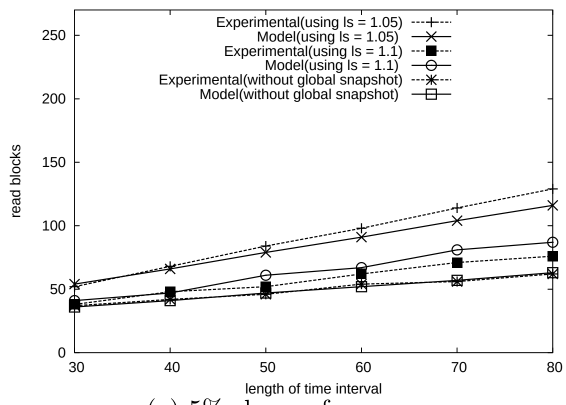


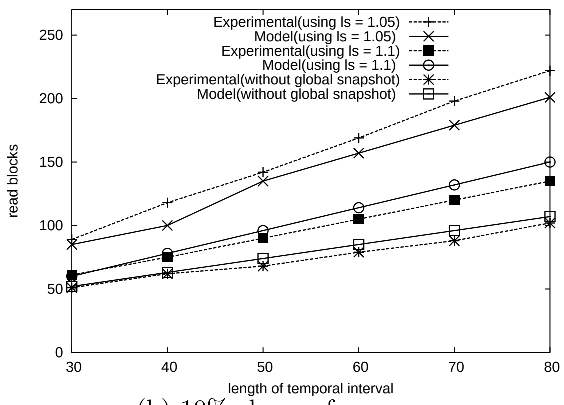
Figure 11: Estimation of the time cost of LES-tree for the data set NUD (change frequency = 10%, and 6% in each dimension of the spatial window)

5 Experimental evaluation

This section presents an experimental evaluation that compares LES-tree with respect to SEST-Index and MVR-tree. The evaluation considers two scenarios with the same number of objects (23,268 points) and time instants stored in the database. While in the first scenario objects initially distribute



(a) 5% change frequency



(b) 10% change frequency

Figure 8: Estimation of the efficiency of the queries processed with LES-tree and uniform distribution (spatial range of the queries made up of 6% of each dimension).

uniformly over the space, in the second scenario objects present the non-uniform initial distribution shown in Figure 10.

5.1 Evaluation of LES-tree with uniform distribution of objects

All experiments in this section consider a density threshold value of 30% to create new global snapshots. With this value, we create only one global snapshot. A stricter value of this density threshold (i.e., a value less than 30%) creates unnecessary global snapshots. As we will see later, the advantage of using global snapshots is reflected in the evaluation of LES-tree with data of a non-uniform distribution of objects (see Figure 10).

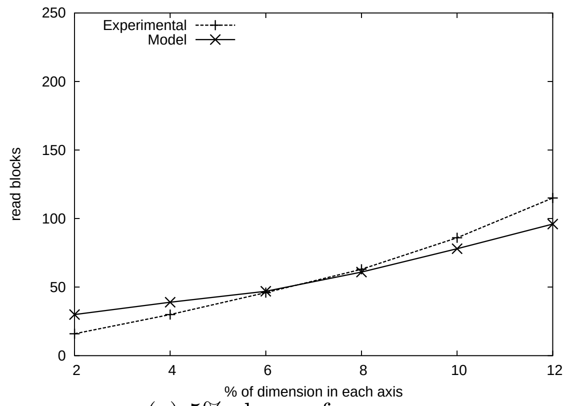
A first experimental evaluation compares LES-tree with SEST-Index and MVR-tree. It compares storage and time costs of all methods when processing spatio-temporal queries. After the initial uniform distribution of objects, objects change their positions over 200 time instants. The set of objects was obtained with the spatio-temporal data generator GSTD [19].

The evaluation uses different lengths of the *logs* that store events, that is, different values of parameter d (see Figure 3) for the structures SEST-Index and LES-tree. Such lengths are determined in terms of the number of events that occur in 1, 12 and 24 time instants. For example, if we consider the number of events that occur at 1 time instant for 23,268 objects with 10% of change frequency, the value of d is 1 for LES-tree and 56 for SEST-Index. Note that by keeping *logs* that store only the events occurr at 1 time instant, both LES-tree and SEST-Index reach the best performance for query processing, to the price of the largest storage cost.

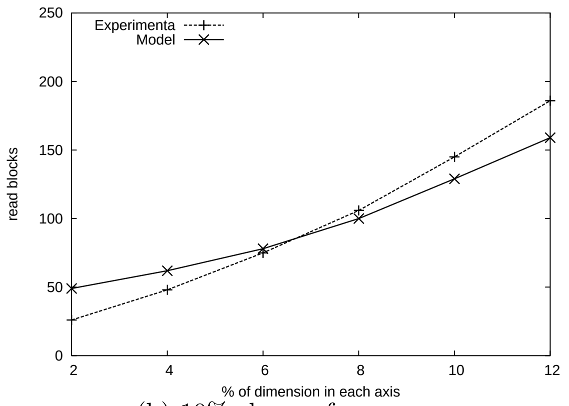
For SEST-Index, the value d is obtained with the expression $d = \frac{p \cdot N \cdot k}{c}$, where k represents the number of time instants (i.e., 1,12 or 24) for which changes are stored in the *logs*. Likewise, for LES-tree the value d is obtained by the equation $d = \frac{p \cdot f \cdot k}{c}$, where d, N, c and f are defined in Figure 3.

5.1.1 Storage cost

Figure 12 shows the storage cost of MVR-tree, SEST-Index and LES-tree for different values of change frequency. In the case of LES-tree and SEST-Index, as d increases, the storage cost reduces. In Figure 12 we can observe that LES-tree requires approximately the same storage than MVRT-tree with a parameter d

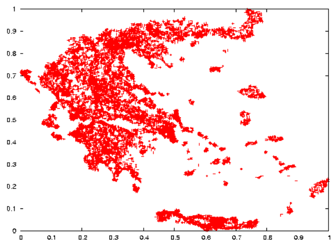


(a) 5% change frequency

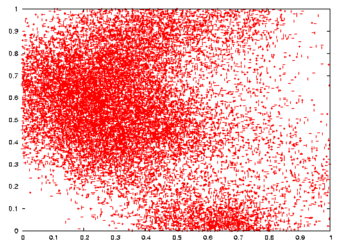


(b) 10% change frequency

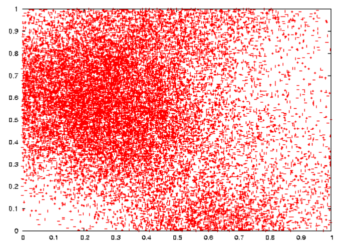
Figure 9: Estimation of the efficiency of the queries processed with LES-tree considering uniform distribution (length of temporal interval equal to 40 units and $ls = 1.1$).



(a)



(b)



(c)

Figure 10: Data of moving objects: (a) instant 1, (b) instant 100 and (c) instant 199

for storing 1 time instant. Remember that, in terms of storage cost, this is the worst scenario for LES-tree. LES-tree, however, overcomes MVR-tree with increasing values of d. For example, if we consider 10% change frequency and values of d equal to 1 and 4, LES-tree requires, approximately, 78% and 65% of the storage of MVR-tree, respectively. For the same change frequency and values of d equal to 560 and 1,120, SEST-Index requires less storage than MVR-tree and LES-tree; however, and as we will show later, the performance of SEST-Index is worse than the performance of LES-tree and MVR-tree.

5.1.2 *Time-slice* and *time-interval* queries

Figures 13, 14 and 15 show the performance of MVR-tree, SEST-Index and LES-tree to process spatio-temporal queries $(Q, [t_i, t_f)$ that consider different lengths for time intervals $[t_i, t_f]$ and different areas for Q. In Figure 13 we can see that LES-tree overcomes by far SEST-Index. Note that this figure only shows the best scenarios of SEST-Index, that is, $d = 56$ for 10% change frequency and $d = 25$ for 5% change frequency, and where its storage is larger than the storage of LES-tree (for $d = 1, d = 2$ y $d = 4$) and MVR-tree.

The unfavorable performance of the SEST-Index

change frequency (%)	MVR-tree / MV3R-tree (Mb)	LES-tree						SEST-Index					
		1		12		24		1		12		24	
		d	(Mb)	d	(Mb)	d	(Mb)	d	(Mb)	d	(Mb)	d	(Mb)
1	6 / 6.3	1	5	1	5	1	5	6	55	55	10	110	6
5	26 / 26.5	1	19	1	19	2	16	25	59	270	14	540	10
10	46 / 46.8	1	36	2	29	4	24	56	64	560	19	1120	15
15	66	1	55	3	38	6	35	75	68	750	25	1500	20
20	81	1	74	4	47	8	43	102	73	1000	30	2000	25
25	101	1	94	5	56	10	52	126	78	1250	35	2500	30

Figure 12: Storage space for MV-tree, LES-tree and SEST-Index (d is the number of blocks used to store changes)

with respect to LES-tree is explained by two factors:

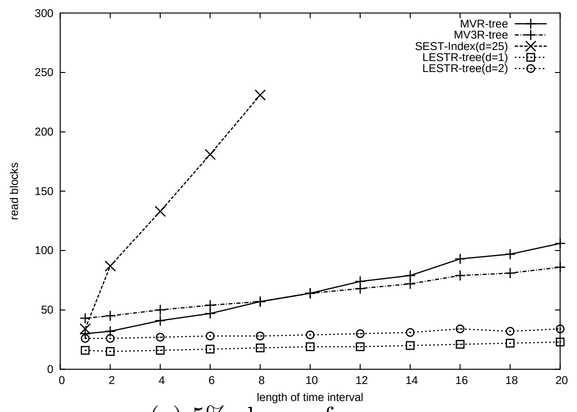
i) The SEST-Index duplicates all the objects in each snapshot including those without changes. The problem of duplication could, in principle, be solved by using the overlapping strategy of HR-tree, but the experimental results showed that there is a low saving in storage, around 5% – 7%. Instead, LES-tree builds a new global snapshot only for the leaves that have undergone several changes.

ii) The SEST-Index groups the changes only in relation to time, and not in relation to time and space as does LES-tree. All changes that occur between consecutive leaf snapshots are stored in a single *log*, which may span many disk blocks. In a query (Q, t), Q is used only for getting the initial set of objects, but not for filtering the *log* blocks in the subsequent processing. Thus, all events between the time of the leaf snapshot and the query time t are processed. This can be alleviated by enforcing short *logs* (with a small d), but this exacerbates problem a).

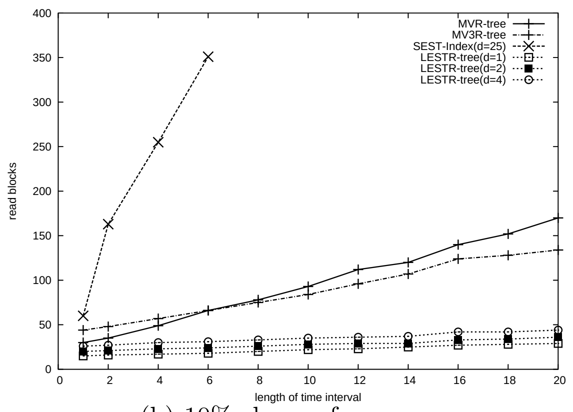
In Figure 13, LES-tree overcomes MVR-tree for all values of d considering 5% and 10% change frequency. Even for the worst scenarios of LES-tree to evaluate queries ($d = 2$ for 5% frequency and $d = 4$ for 10% change frequency), the performance of LES-tree is better than the performance of MVR-tree, a difference that increases with the length of the query time interval. Recall that for these scenarios, LES-tree requires less storage than MVR-tree.

Figures 14 and 15 show the behavior of MVR-tree, SEST-Index and LES-tree for *time-interval* queries for which the temporal interval lengths were 1 and 4, and the spatial range was made up of 2% – 20% from each dimension. In Figure 14 (*time-slice* query) with $d = 1$, LES-tree overcomes MVR-tree for all the areas of query window. In this scenario, both structures use approximately the same storage. In the same figure, SEST-index overcomes LES-tree as the area of the query window increases. For example, in Figure 14-a (5% change frequency), SEST-Index starts to outperform LES-tree from an area made up of 14% of each dimension for $d = 1$. In contrast, for 10% change frequency, SEST-Index presents better performance than the performance of LES-tree starting from areas covering 20% of each dimension. SEST-Index, however, requires 3 times and twice the space than LES-tree for change frequency of 5% ($d = 24$) and 10% ($d = 56$), respectively. The experiment also shows that the area of the queries when SEST-Index outperforms LES-tree increases as the temporal interval length enlarge. The latter can be seen in Figure 15 where LES-tree overcomes easily MVR-tree and SEST-Index for all areas of query window and values of $d = 1$ and $d = 2$ for LES-tree.

There exists a spatio-temporal access method known as MV3R-tree [15] that corresponds to a



(a) 5% change frecuency



(b) 10% change frecuency

Figure 13: Blocks read by MVR-tree, SEST-Index and LES-tree for queries with different temporal interval lengths (10% change frequency and spatial range made up of 6% of the length of each dimension).

variant of MVR-tree. MV3R-tree combines MVR-tree with a small 3D R-tree[1][15]. MVR-tree solves queries with short temporal intervals, whereas 3D R-tree solves queries with long temporal intervals. In [15], MV3R-tree, HR-tree [10, 11], and the original 3D R-tree are compared[2][21]. The results show that the 3D R-tree used by MV3R-tree requires only 3% of the storage used by MVR-tree (thus MV3R-tree is only 3% larger than MVR-tree). MV3R-tree uses 1.5 times the storage required by the original 3D R-tree, and MV3R-tree needs a minimum fraction of the HR-tree storage. On the other hand, our experiments show that LES-tree requires, in average, approximately 75% of the storage used by MVR-tree, considering all changes occurred in 12 time instants. Thus, we can state that LES-tree needs approximately the same storage than 3D R-tree.

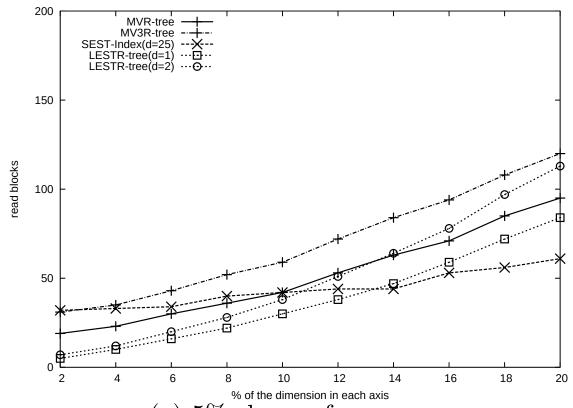
With respect to the queries with short temporal intervals (less than 5% of the total of the temporal intervals of the database), MV3R-tree uses a MVR-tree to evaluate them. In our experiments, data contains 200 snapshots and, therefore, LES-tree will outperform the MV3R-tree in the same situations where we considered temporal intervals less than or equal to 10 time units. For queries that consider a temporal interval whose length is larger than 5%, MVR-tree needs to do approximately between 75% and 80% of the accesses required by MVR-tree as is shown in [13]. Assuming that the cost (time) of MV3R-tree will be 75% of MVR-tree, it is possible to affirm that LES-tree is always better than MV3R-tree for queries whose temporal interval is larger than 2 time units (for $d = 4$ and change frequency of 10% representing the worst scenario of LES-tree).

It is also possible to conclude that LES-tree is better than the spatio-temporal access method that uses the original 3D R-tree, given that the results in [15] show that 3D R-tree does not outperform MV3R-tree in none of the analyzed situations. In queries with long temporal intervals, 3D R-tree experimented a performance very similar to that of MV3R-tree.

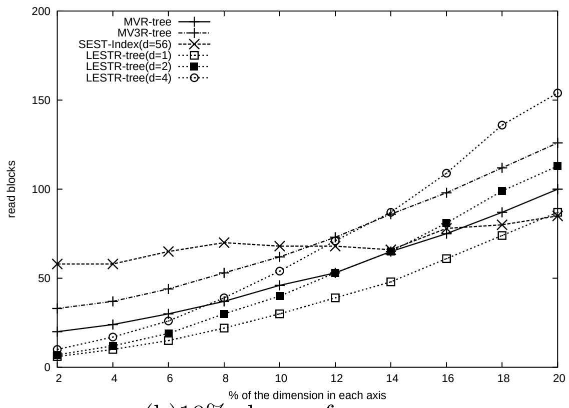
LES-tree is an event-oriented access method that aims to efficiently process not only *time-slice* and *time-interval* queries, but also queries on *events*. To guarantee performance for all queries, the structure maintains some attributes that can be eliminated if we are only interested in *time-slice* or *time-*

[1]This 3D R-tree is created considering the leaves of a MVR-tree; that is, each leaf of the MVR-tree is an object in the 3D R-tree

[2]In constrast, this 3D R-tree stores each object instead of each leaf of the MVR-tree



(a) 5% change frecuency



(b)10% change frecuency

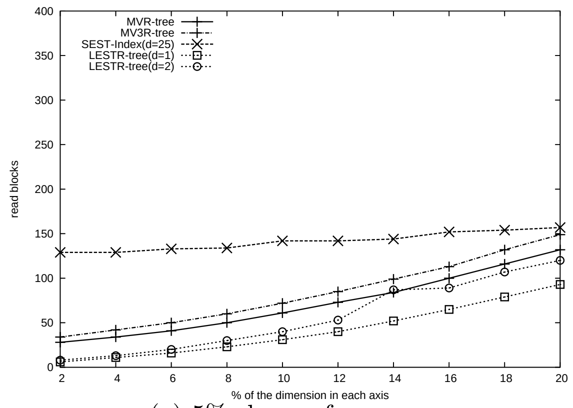
Figure 14: Blocks read by MVR-tree, SEST-Index and LES-tree for *time-slice* queries (temporal interval length = 1) in different spatial ranges.

interval queries. More specifically, it is possible to eliminate the *Geometry* attribute in the *move_out* type event and still achieve a better performance in the processing of *time-slice* and *time-interval* queries. We evaluated the performance of LES-tree considering these modifications in the data structure (adjusted LES-tree). The results show that for 10% change frequency and $d = 4$, LES-tree requires approximately 67% of the storage required by the original LES-tree and 70% of the time to process queries whose query space window covers 6% in each dimension and whose lengths of time intervals go from 1 to 20 units.

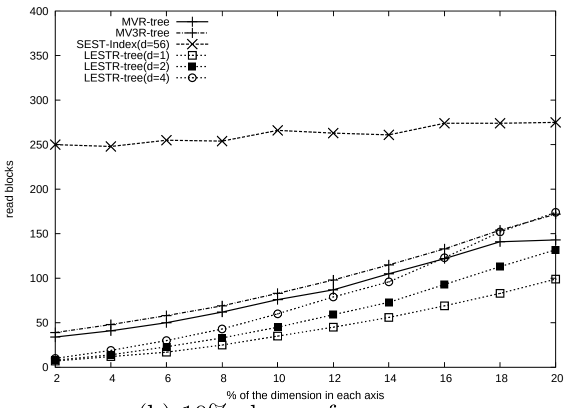
5.1.3 Queries on *events*

As we explained above, SEST-Index and LES-tree also enable to process queries about *events* that occurred in time instants or time intervals. The processing cost of this type of queries with LES-tree is the same as the one needed for the *time-slice* and *time-interval* queries. For SEST-Index, however, it is possible to have advantages to process event queries. We evaluate both structures with 10% change frequency and parameter d such that both structures require the same storage. To fulfill this constraint, d was set to 816 and 4 for SEST-Index and LES-tree, respectively. In this scenario SEST-Index overcomes LES-tree when the query window covers an area formed by more than 11% of each dimension. SEST-Index, however, presents a poor performance for *time-slice* and *time-interval* queries with respect to LES-tree. For example, a *time-slice* query with LES-tree requires approximately 8% of the time needed with SEST-Index.

Despite the fact that MVR-tree does not provide an algorithm to solve queries about *events*, it may be possible to process *events* queries. For example, an event query can be “find how many objects enter a region (Q) at a time instant t.” To process this query with MVR-tree, two *time-slice* queries are needed: one at time instant t and the other one at the instant immediately before t (t'). Objects that enter Q at t correspond to those that were not found in Q in t', but are now found in Q at t. The cost of processing this query with MVR-tree has the double cost of processing a *time-slice* query. Indeed, for the above query with a region covering 6% of each dimension, 10% change frequency, and $d = 4$, LES-tree requires access only 43% of the nodes or blocks accessed by MVR-tree.



(a) 5% change frecuency



(b) 10% change frecuency

Figure 15: Blocks read by MVR-tree, SEST-Index and LES-tree for queries in different spatial ranges (temporal interval length = 4).

5.2 Evaluation of LES-tree with non-uniform data distribution

This section compares LES-tree with global snapshots against MVR-tree by using NUD data set (defined in Section 4.2.3). To obtain the storage and time costs of queries, we use the implementation of MVR-tree and LES-tree, this last with $d = 4$.

Figure 16 shows the values of density $realDen$ under five different scenarios: (1) the density obtained with LES-tree, 23,268 objects (points), and an initial uniform distribution; (2) the density of LES-tree with the set of objects NUD without global snapshots; (3), (4) and (5) densities of LES-tree when considering the set of objects NUD, $li = 1$, and thresholds equal to $ls = 1.3$, $ls = 1.6$, and $ls = 1.8$, respectively.

The space used by LES-tree when considering thresholds $ls = 1.3$, $ls = 1.6$, and $ls = 1.8$ was 33Mb, 29Mb, and 28Mb, respectively, against 46 Mb required by MVR-tree and 24 Mb required by the LES-tree without global snapshots. Figure 16 indicates that, when considering a threshold $ls = 1.6$, two global snapshots are created (approximately at time instants 10 and 50). This produces an important improvement on the density, reaching values that are similar to those of scenario (1). Similar results are obtained with a threshold $ls = 1.3$. With a threshold $ls = 1.8$ in scenario (5), however, the density continues being larger than in scenario (1), affecting negatively the time cost of LES-tree.

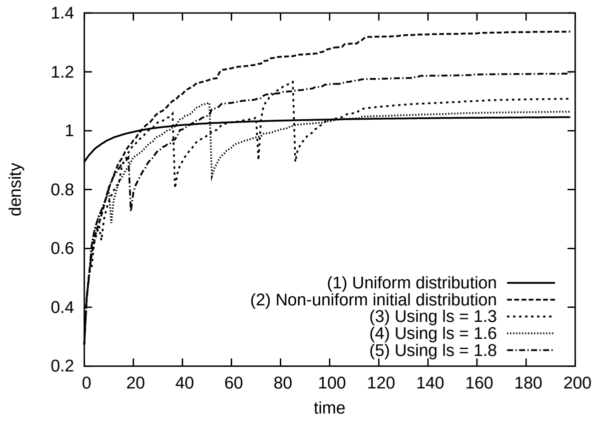


Figure 16: Density for *logs* of LES-tree

Figure 17 shows the query performance of MVR-tree and LES-tree in scenarios (1) to (5) (in addition to the performance of MVR-tree using NUD data).

In this Figure it is possible to observe that with $ls =$

1.3 or $ls = 1.6$, LES-tree, with global snapshots, shows a similar performance that the LES-tree without global snapshots, considering objects (23,268 points) with uniform distribution. Also, LES-tree with $ls = 1.3$ or $ls = 1.6$ overcomes MVR-tree in almost all time intervals evaluated. The advantage of LES-tree over MVR-tree in this scenario (non uniform distribution) to process queries increases drastically as the value of d decreases and we use the adjusted LES-tree as described before.

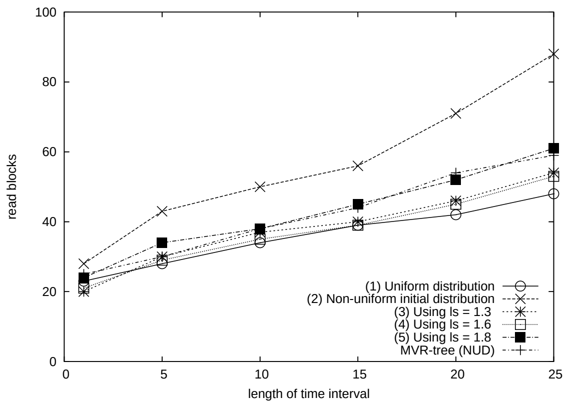


Figure 17: Query performance of LES-tree with global snapshots and MVR-tree over the set of objects con distibución uniforme y no uniforme (NUD)

6 Conclusion and future work

This work proposes a new spatio-temporal access method, LES-tree, that handles events and snapshots associated with space partitions. Based on the experimental results, LES-tree (with parameter $d = 4$) requires around 64% of the space used by MVR-tree, the best alternative structure up to date. On the other hand, LES-tree outperforms the MVR-tree for *time-slice* and *time-interval* queries. Unlike other proposed spatio-temporal access methods, it is also possible to process queries about *events* using LES-tree with a similar efficiency as the algorithms used to process *time-slice* and *time-interval* queries. In this paper we also described and validated a cost model for LES-tree. This model enables to estimate the storage and the efficiency of the queries processed with LES-tree (with and without global snapshots), and with a relative average error of about 15% for storage and 11% for queries.

Besides traditional *time-slice* and *time-interval*, as well as queries on *events*, LES-tree 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 [7], called spatio-temporal pattern queries (STP). LES-tree 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. Algorithms and experimental evaluations for these type of queries have been left out of the scope of this paper.

In addition, we are developing a method to obtain the values of the parameters for LES-tree such that the structure is optimized with respect to pre-defined constraints for storage or time cost. We also plan to include in the cost model the effect of using a buffer for caching blocks or pages of the structure. Finally, we are studying new algorithms for joins and nearest-neighbor queries.

References

[1] Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion B-tree. The VLDB Journal **5**(4), 264–275 (1996). DOI http://dx.doi.org/10.1007/s007780050028

[2] Cole, S.J., Hornsby, K.: Modeling noteworthy events in a geospatial domain. In: M.A. Rodríguez, I.F. Cruz, M.J. Egenhofer, S. Levashkin (eds.) GeoSpatial Semantics, First International Conference, GeoS, 2005, Mexico City, Mexico, November 29-30, 2005, Proceedings, *Lecture Notes in Computer Science*, vol. 3799, pp. 77–89. Springer (2005)

[3] Galton, A., Worboys, M.F.: Processes and events in dynamic geo-networks. In: M.A.

Rodríguez, I.F. Cruz, M.J. Egenhofer, S. Levashkin (eds.) GeoSpatial Semantics, First International Conference, GeoS, 2005, Mexico City, Mexico, November 29-30, 2005, Proceedings, *Lecture Notes in Computer Science*, vol. 3799, pp. 45–59. Springer (2005)

[4] Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques and applications. IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing **18**(2), 3–18 (1995). URL citeseer.ist.psu.edu/gupta95maintenance.html

[5] Gutiérrez, G., Navarro, G., Rodríguez, A., González, A., 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), pp. 115–124. ACM Press (2005)

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

[7] Hadjieleftheriou, M., Kollios, G., Bakalov, P., Tsotras, V.J.: Complex spatio-temporal pattern queries. In: Proceedings of the 31st international conference on Very large data bases (VLDB '05), pp. 877–888. VLDB Endowment (2005)

[8] Kollios, G., Tsotras, V.J., Gunopulos, D., Delis, A., Hadjieleftheriou, M.: Indexing animated objects using spatio-temporal access methods. Knowledge and Data Engineering **13**(5), 758–777 (2001). URL citeseer.nj.nec.com/494812.html

[9] Kollios, G.N.: Indexing problems in spatiotemporal databases. Ph.D. thesis, Polytechnic University, New York (2000)

[10] Nascimento, M.A., Silva, J.R.O., Theodoridis, Y.: Access structures for moving points. Tech. Rep. TR–33, TIME CENTER (1998). URL citeseer.nj.nec.com/article/nascimento98access.html

[11] Nascimento, M.A., Silva, J.R.O., Theodoridis, Y.: Evaluation of access structures for discretely moving points. In: Proceedings of the International Workshop on Spatio-Temporal Database Management (STDBM '99), pp. 171–188. Springer-Verlag, London, UK (1999)

[12] Pfoser, D., Tryfona, N.: Requirements, definitions, and notations for spatio-temporal application environments. In: Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems (GIS'98), pp. 124–130. ACM Press (1998). DOI http://doi.acm.org/10.1145/288692.288715

[13] Tao, Y., Papadias, D.: MV3R-tree: A spatio-temporal access method for timestamp and interval queries. Tech. Rep. HKRUST-CS00-06, Department of Computer Science, Hong Kong University of Science Technology, Hon Kong (2000)

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

[15] Tao, Y., Papadias, D.: MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In: Proceedings of the 27th International Conference on Very Large Data Bases, pp. 431–440. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)

[16] Tao, Y., Papadias, D., Zhang, J.: Cost models for overlapping and multiversion structures. ACM Trans. Database Syst. **27**(3), 299–342 (2002). DOI http://doi.acm.org/10.1145/581751.581754

[17] Theodoridis, Y., Sellis, T.: A model for the prediction of R-tree performance. In: Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS '96), pp. 161–171. ACM Press, New York, NY, USA (1996). DOI http://doi.acm.org/10.1145/237661.237705

[18] Theodoridis, Y., Sellis, T.K., Papadopoulos, A., Manolopoulos, Y.: Specifications for efficient indexing in spatiotemporal databases. In: IEEE Proceedings of the 10th International Conference on Scientific and Statistical Database Management, pp. 123–132 (1998). URL citeseer.nj.nec.com/theodoridis98specifications.html

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

[20] Theodoridis, Y., Stefanakis, E., Sellis, T.: Efficient cost models for spatial queries using R-Trees. IEEE Transactions on Knowledge and Data Engineering **12**(1), 19–32 (2000). DOI http://dx.doi.org/10.1109/69.842247

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

[22] Tzouramanis, T., Vassilakopoulos, M., Manolopoulos, Y.: Overlapping linear quadtrees and spatio-temporal query processing. The Computer Journal **43**(4), 325–343 (2000). URL citeseer.nj.nec.com/tzouramanis00overlapping.html

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

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