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

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

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

Andrea Rodríguez T.
Universidad de Concepción
Center for Web Research
Edmundo Larenas 215, Concepción / Chile
andrea@udec.cl

Alejandro González O.
Universidad del Bío-Bío
Avenida La Castilla S/N - Chillán / Chile
alejandro.gonzalez@dmr-consulting.int

José Orellana V.
Universidad del Bío-Bío
Avenida La Castilla S/N - Chillán / Chile
jose.orellana@gmail.com

ABSTRACT

This paper describes a new spatio-temporal access method (SEST-Index) that combines two approaches for modeling spatio-temporal information: snapshots and events. This method makes it possible to not only process *time slice* and *interval* queries, but also queries about events. The SEST-Index implementation uses an R-tree structure for storing snapshots and a log data structure for storing events that occur between consecutive snapshots. Experimental results that compare SEST-Index and HR-tree show that, for a change frequency between 1% and 13%, SEST-Index requires less storage space than HR-tree, and for a change frequency between 1% and 7%, SEST-Index outperforms HR-tree for *interval* queries. In addition, as SEST-Index is an event-oriented structure, event queries are efficiently answered. In order to decrease the storage space for frequencies of change above 20%, this work explores alternatives that optimize the space of the log structure without affecting the efficiency of query answers.

[*]Gilberto Gutiérrez is partially funded by Dirección de Investigación, Universidad del Bío-Bío, Grant 043318 3/R. Gonzalo Navarro and Andrea Rodríguez are funded by Nucleus Millenium Center for Web Research, Grant P04-067-F, Mideplan, Chile

Categories and Subject Descriptors

H.2 [**DATABASE MANAGEMENT**]: Database applications - *Spatio-temporal databases*

General Terms

Algorithms, Performance

Keywords

Spatio-temporal access methods, R-trees, temporal events.

1. INTRODUCTION

Space and time are two inherent attributes of any object of the real world. Thus, an object is characterized by its position and extent at any instant in time [4]. These objects make up the type of spatio-temporal data that need to be managed in some computer applications. For example, an application of a fleet of taxis needs to store information about where and when each of its cars has been. This allows us to answer queries such as, *Which were the cars located at the shopping center at 6:00 pm ?* or *Which are the closest cars to the car number BB-3545 (which needs assistance)?*. Other applications relate to transportation, environment, social (e.g., demographic and health) and multimedia systems. Spatio-temporal applications have been classified into three categories depending on the type of data they manage [8].

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

2. Applications that include objects located in space and that can change their position by means of a modification of their geometric shape. For example, the

changes in the administrative boundaries of a city over time. In this type of applications the changes in the geometric shape of the objects occur in a discrete manner.

3. Applications that integrate both previous behaviors. This type of applications appears in the environmental area where it is necessary to model the movements of the objects and their geometric shapes in time.

This work proposes a new access method (SEST-Index) that is adequate for applications that belong to the second category, thus it supports discrete changes to the location and shape of the objects. This access method is based on producing snapshots after a certain number of changes that occur over objects and on storing the events that produce these changes in a data structure called *log*. Consequently, it allows the representation of (1) *temporal snapshots* and (2) *events on objects*, described in [16]. This approach has been discussed in others studies [2, 3], but it has been discarded a priori by arguing that it is not easy to determine how many events determine a new snapshot and that extra time is required for query processing. The number of snapshots represents a tradeoff 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. This work explores and experimentally evaluates the combination of snapshots and events for the following reasons:

1. Both snapshots and events are considered complementary and relevant information for spatio-temporal applications. Interesting queries exist for objects' states and for events over objects. For example, *when did an object enter a region?* and *how many objects move out of a region within a given time interval?*.

2. 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.

3. The data structure for snapshots and changes or events are independent, and so are the improvements that can be obtained in either structure. Moreover, integration of existing spatial access methods for handling snapshots into this approach can be easily achieved.

There exist various spatio-temporal access methods for this same category of applications. Some are RT-tree [17], HR-tree (Historical R-tree) [6, 7], 3D R-tree [14], HR$^+$-tree [10], MV3R-tree [11] and OLQ (Overlapping Linear Quadtree)[15], among others that are designed to only answer *time slice* and *time interval* queries about the history of the special attributes of objects. Unlike these previous studies, this work aims to define a new access method that can efficiently answer time slice, time interval and event-based queriers.

The organization of this article is as follows. In Section 2 the advantages and disadvantages of the main spatio-temporal access methods are discussed. Section 3 describes the proposed access method, considering data structures and algorithms for query processing and updates. In Section 4 an experimental evaluation is presented. Section 5 presents some variants of SEST-Index. Finally, Section 6 presents conclusions and future work.

2. SPATIO-TEMPORAL ACCESS METHODS

This section describes the main spatio-temporal access methods available for applications of category 2 (Section 1).

Figure 1 shows an example of the evolution of a set of spatio-temporal objects in different instants of time. For simplicity, an example in a two-dimensional space is considered. In Figure 1, the axes x and y represent the two-dimensional space while t corresponds to the temporal dimension. In instant of time t_1, objects O_1 and O_2 are inserted. In instant t_2, object O_3 is inserted while object O_1 moves and O_2 changes its shape. In instant t_5, object O_1 moves again and object O_2 changes its form until it completely disappears. A *time slice* query is shown in Figure 1. This query is expressed in the following way: *find objects that appear in the rectangle q at instant t_3*.

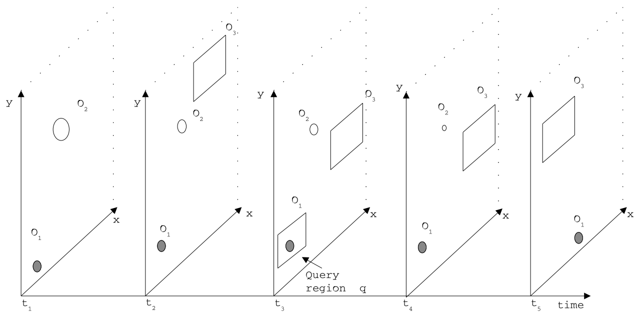

Figure 1: An example of the evolution of spatio-temporal objects

According to [12] and [5], it is possible to classify the spatio-temporal access methods into the following three categories:

- Methods that treat time as another dimension.
- Methods that incorporate the temporal information within the node structure without considering time as another dimension.
- Methods based on overlapping and multiversion of the structure. In this case, the temporal dimension is separated from the spatial dimension.

The 3D R-tree [14] considers time as another axis along with the spatial coordinates. Using this approach, an object that initially remains at (x_i, y_i) during time interval $[t_i, t_j)$ and then at (x_j, y_j) during time interval $[t_j, t_k)$ can be modeled by two line segments in a three-dimensional space $[(x_i, y_i, t_i), (x_i, y_i, t_j))$ and $[(x_j, y_j, t_j), (x_j, y_j, t_k))$, which 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. A disadvantage of this approach is the inefficiency of processing time slice queries. Advantages of this approach are the efficiency in the use of space and the efficiency in processing time interval queries.

RT-tree [17] is a structure belonging to the second category. In this structure, the temporal information is kept in the nodes of the R-tree. This is an extension of the information content that an R-tree normally has. 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 [6].

HR-tree [7, 6] and MR-tree [17] belong to the third category. Both 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. HR-tree is one of the most studied methods for which evaluations have been made and for which variants exist. 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 in instant t_i, the tree is completely duplicated at time instant t_{i+1}. MV3R-tree [11] also belongs to the third category, using a multiversion approach. MV3R-tree uses two structures: a MVR-tree (Multi-version R-tree) [11] for processing timeslice queries (where HR-tree has advantage) and an auxiliary 3D R-tree for processing long interval queries (where 3D R-tree has advantage).

3. PROPOSED METHOD: SEST-INDEX

The idea behind our method consists in maintaining the snapshots of the database for certain instants of time and a log to store the events occurred between consecutive snapshots. When an object undergoes a change in its spatial attribute at a given time instant, it generates a change event. The log is store in time-order and allows us to reconstruct whatever the state of the database was between two consecutive snapshots (see Figure 2). The proposed access method considers that, for each snapshot, the spatial components of the live objects in the database are stored in an R-tree data structure. As it was explained previously, changes are maintained in the log. For example, in Figure 2 the state of the objects in the 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 in instant t_0 and update objects' attributes (i.e., location) with the information of log L_0

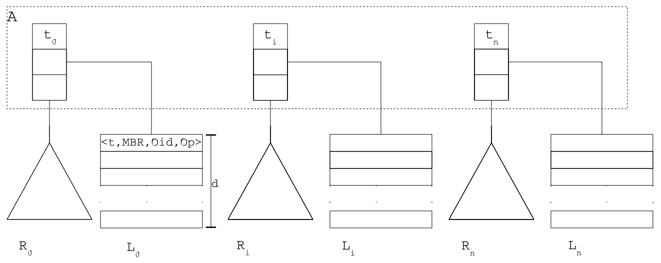

Figure 2: General outline of SEST-Index

3.1 Structure description

The structure of the R-tree is the same as the proposal in [1], and the data structure of the log is a linked list of blocks. The entries in the blocks are tuples with the following structure: $< t, Geometry, Oid, Op >$, where t corresponds to the time in which the modification (for example the insertion of a new object) happened, and *Oid* is the object identifier. *Geometry* corresponds to the values in the spatial component of the object, which depends on the type of spatial objects that need to be stored. For example, if the objects are points in a two-dimensional space, then *Geometry* will be a pair of coordinates (x, y). If the objects are polygons, then *Geometry* will be a set of points that defines the polygon or its approximation, for example, the polygon's MBR (Minimum Bounding Rectangle). Finally, *Op* indicates the type of operation (i.e., type of event or type of modification). For this work,we consider just two types of operations: *move_in* (an object moves to a new position) and *move_out* (an object leaves its current position). Thus an object creation is modeled as a *move_in*, an object deletion as a *move_out*, and an object changing its position or its shape as a *move_out* followed by a *move_in*. The entries in the log are ordered by the attribute t.

The whole index is a data structure A, which is a sequence of snapshots A_i such that $A_i.t$ is the corresponding time instant of the i-th snapshot, $A_i.R$ is the R-tree for such snapshot, and $A_i.L$ is the log of events between time instants $A_i.t$ and $A_{i+1}.t$. A parameter d defines the maximum allowed distance (in disk blocks or changes) between consecutive snapshots.

3.2 Operations

3.2.1 Time slice queries

To process a time slice query, the first step is to find the corresponding snapshot according to the time instant t that is specified in the query. This snapshot corresponds to the latest time instant within $[0, t]$ when a snapshot has been stored. Using this snapshot, a spatial search is done using the method defined in [1] for an R-tree. The set of objects obtained from this query is updated with the entries in the corresponding log (Figure 3).

```
 1: TimeSliceQuery(Rectangle $q$, Time $t$)
 2: {$R$ will be a set that stores the objects of the answer}
 3: Find the last entry $i$ in $A$ such that $A_i.t \leq t$
 4: $R$ = SearchRtree($A_i.R, q$) {All the objects in $A_i.R$ that intersect $q$ at instant $A_i.t$ are retrieved}
 5: for each entry $e \in A_i.L$ so that $e.t \leq t$ do
 6:   if $e.Geometry\ Intersect(q)$ then
 7:     if $e.Op = Move\_in$ then
 8:       $R = R \cup \{e.Oid\}$
 9:     else
10:       $R = R - \{e.Oid\}$
11:     end if
12:   end if
13: end for
14: return $R$
```

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

3.2.2 Time interval query

Similar to the process for time slice queries, objects from the R-tree are retrieved by starting at a previous reference point that is closest to the starting instant t_1 of the time interval of the query. Then, the process continues with all entries in the log whose time instant is less than or equal to the upper bound t_2 of the time interval of the query (Figure 4).

```
1: IntervalQuery(Rectangle $q$, Time $t_1$, Time $t_2$)
2: $G = \emptyset$ {$G$ is a set that stores objects of the answer}
3: Search last entry $i$ in $A$ such that $A_i.t \leq t_1$
4: $R$ = SearchRtree($A_i.R, q$) {All objects of $A_i.R$ that intersect
   $q$ in the instant $A_i.t$ are retrieved}
5: $L = A_i.L$
6: $k = i$
7: Update $R$ with the changes in the log $L$ which are found in
   the interval $[t, t_1]$ (just like a TimeSliceQuery)
8: $G = R$
9: if all the entries in log $L$ were processed then
10:    $k = k + 1$
11:    $L = A_k.L$
12: end if
13: Let $t_s$ be the next instant after $t_1$ stored in Log $L$
14: while $t_s \leq t_2 \wedge$ entries remain to be processed in $L$ do
15:    Update $R$ with the changes in Log $L$ ocurred in $t_s$
16:    $G = G \cup R$
17:    if all the entries in log $L$ were processed then
18:       $k = k + 1$
19:       $L = A_k.L$
20:    end if
21:    Let $t_s$ be the next instant stored in Log $L$
22: end while
23: return $G$
```

Figure 4: Algorithm to process an *Interval* type query

3.2.3 Queries about events

With SEST-Index it is possible to process queries about events. For example, given an area q and a time t, find how many objects *move_in* the area q and how many objects *move_out* the area q at time instant t. A simple implementation of this type of queries consists in using an array B to locate the log block *Bid* containing the events occurred at time t. Each entry of the array B occupies a small space and, therefore, it is possible to keep various entries in a disk block. For example, for blocks of size 1024 bytes, it is possible to store approximately 128 entries. Notice that this implementation doesn't need to access an R-tree for processing a query about events. Figure 5 describes the algorithm.

3.2.4 Updating the structure

This method aims to update the index with the changes over objects occurred in a particular time instant. Assume we have a list with all changes that have occurred at a time instant and with all objects that can be found "live" in the database just before this time instant. What the algorithm does is to update the "live" objects at the new time instant that is being inserted in the database. It then verifies if the parameter d satisfies the threshold condition, that is, if the size (in blocks) defined for the logs is reached by the new changes. In such case, a new snapshot is created, which implies creating a new R-tree and a new entry in A. If d has not been reached, the changes are simply stored in the log. Figure 6 describes this algorithm.

4. EXPERIMENTAL EVALUATION

With the aim of evaluating the performance of SEST-Index, it is compared with HR-tree in terms of disk usage

```
1: EventQuery(Rectangle $q$, Time $t$, Array $B$)
2: Search the entry $i$ in array $B$ such that $B_i.t = t$
3: $L = B_i.Bid$
4: $terminated = false$
5: $oi = 0$ {quantity of objects that entered $q$ at time $t$ }
6: $os = 0$ {quantity of objects that left $q$ at time $t$ }
7: while not terminated do
8:    for each entry $e \in L$ so that $e.t \leq t$ do
9:       if $e.t = t \wedge e.Geometry\ Intersect(q)$ then
10:         if $e.Op = Move\_in$ then
11:            $oi = oi + 1$
12:         else
13:            $os = os + 1$
14:         end if
15:      end if
16:   end for
17:   if $e.t > t$ then
18:      $terminated = true$
19:   else
20:      $L$ = next log block
21:   end if
22: end while
23: return $(oi, os)$
```

Figure 5: Algorithm to process queries about *events*

```
1: InsertChanges(Time $t$, Changes $c$, SnapShot $F$) {$t$ is
   the instant of time in which the changes happen, $c$ is a list
   with the changes occurred at $t$ and $F$ corresponds to the "live"
   elements in the instant immediately preceding $t$}
2: Let $i$ be the last entry of $A$.
3: Update $F$ with the changes of $c$
4: Insert the elements of $c$ at the end of log $A_i.L$
5: if the total changes in $c$ plus the stored changes in the log
   $A_i.L$ is greater than $d$ then
6:    {Create a new snapshot in SEST-Index} $A_{i+1}.t = t$
      $A_{i+1}.R$ = R-tree with the live objects in $F$ $A_{i+1}.L = \emptyset$
7: end if
```

Figure 6: Algorithm to update the structure

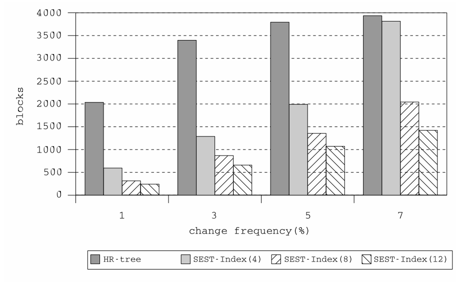

Figure 7: Space usage with 3000 objects

and number of blocks read for the different types of queries described in the previous section (time slice, time interval and event-based queries). In the absence of real data, the experiment uses data obtained with the spatio-temporal data generator GSTD[13]. Sets with 1000, 3000 and 5000 points (objects), respectively, were created. These points are distributed uniformly within a region in time instant 0.0. Subsequently, they are moved randomly during the next 50 time instants until reaching time instant 1.0. Thus, in this evaluation, we consider change events related to the movement of objects. Four percentages of object mobility (change frequency) were considered: 1, 3, 5 and 7 percent per instant of time. Three values for the parameter d were also studied: 4, 8, and 12 disk blocks with a block size of 1024 bytes. Disk usage was measures by the number of blocks used by the data structures after inserting the objects and their changes. Access time was defined as the average number of blocks read for performing 100 random queries.

The tests were performed on a Pentium 4 computer with 1.6 Ghz, 1 Gb of RAM and Linux Operating System.

4.1 Space usage

Figure 7 shows that SEST-Index uses less disk space than HR-tree, and this behavior is more pronounced as the number of objects or the percentage of changes decreases. An extreme case in wich SEST-Index utilizes more disk space than HR-tree occurs when the selected d is so low that SEST-Index is only able to store the changes occurred in a single time instant in each log. In such a case, a new snapshot for each time instant is created and the structure degenerates into many individual R-trees.

A simple idea for reducing even more the space used by SEST-Index is to consider the basic HR-tree strategy, that is, at the instant of creating a new R-tree in a snapshot, reuse part of the R-tree of the previous snapshot. This idea was evaluated, but little space was saved (around 5%).

4.2 Time slice queries

100 queries were performed with rectangles (query windows) formed with 5% and 10% of the dimension in each axis. The results shown for 3000 objects are similar to the results obtained for 1000 and 5000 objects.

We can observe in Figures 8 and 9 that SEST-Index reads more blocks than HR-tree in this type of query. For most cases, SEST-Index needs, on average, twice of the number of disk accesses required by HR-tree. SEST-Index always takes longer to answer to a time slice query than HR-tree, since it does not only read an R-tree but also the corresponding log. Nevertheless, as the parameter d gets smaller, the disadvantages of SEST-Index decrease because small logs imply less blocks read in order to answer a query.

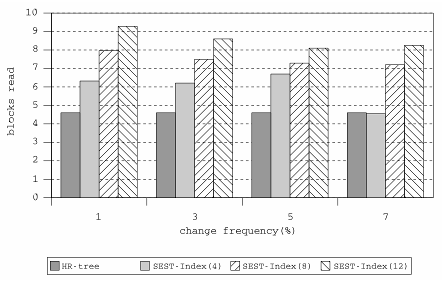

Figure 8: Blocks read by a time slice query with a query window formed by 5% of the dimension in each axis

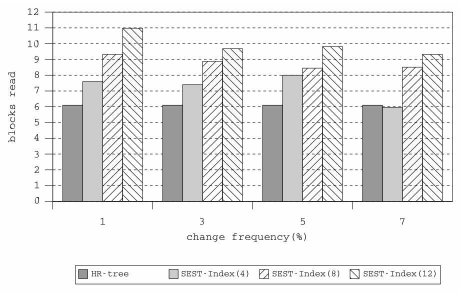

Figure 9: Blocks read by a time slice query with a query window formed by 10% of the dimension in each axis

4.3 Time interval queries

As for time slice queries, the evaluation of time interval queries uses 100 random query windows formed with 5% and 10% of the dimension in each axis. Only results for 3000 objects are shown, since similar conclusions were reached for 1000 and 5000 objects.

We can observe in Figures 10, 11, 12 and 13 that SEST-Index reads less blocks than HR-tree when the change frequency is low (less than 5%). As the query interval grows, the advantage of SEST-Index increases. This advantage is due to the fact that SEST-Index only reads one R-tree and sequentially a log. HR-tree, in contrast, must read an R-tree for each time instant within the query interval.

4.4 Queries about events

In the same way of previous evaluations, 100 queries were performed with rectangles (query windows) formed with 5% and 10% of the dimension in each axis. The results shown for 3000 objects are similar than those obtained for 1000 and 5000 objects. The queries about events were processed using the algorithm in Figure 5. In this algorithm, the number of blocks that were read depends only on the percentage of change frequency and not on the parameter d or the size of the query window. This can be clearly seen in Figure 14.

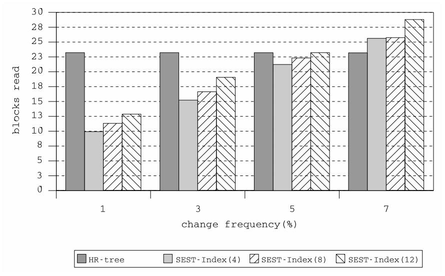

Figure 10: Blocks read in time interval queries with size of time intervals equal to five and query windows formed with 5% of the dimension in each axis

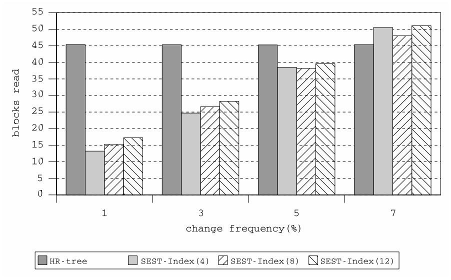

Figure 11: Blocks read in time interval queries with size of time intervals equal to 5 and query windows formed with 10% of the dimension in each axis

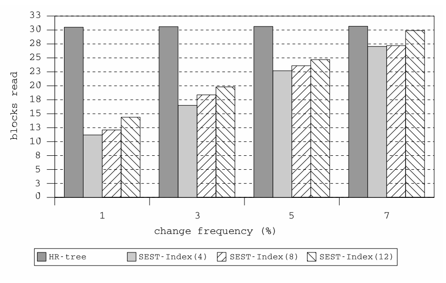

Figure 12: Blocks read in time interval queries with size of time intervals equal to 10 and query windows formed with 5% of the dimension in each axis

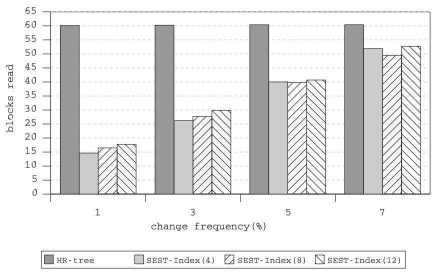

Figure 13: Blocks read in time interval queries with size of time intervals equal to 10 and query windows formed with 10% of the dimension in each axis

Figure 15 shows the number of blocks that were necessary to read for processing queries about events for both SEST-Index and HR-tree, considering 3000 objects and a value d equal to 8 (similar results were obtained for other values of d). We can observe that SEST-Index reads a smaller number of blocks than HR-tree. This is due to the fact that HR-tree needs to query two consecutive instants of time (process two R-trees), whereas SEST-Index makes a direct location of the changes stored in the log.

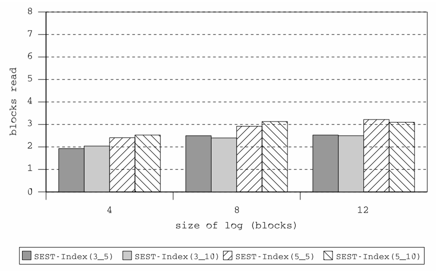

Figure 14: Blocks read by SEST-Index for queries about events for 3000 objects. SEST-Index(i-j) indicates queries for sets of objects with a change frequency of i% and query window formed by j% of the dimension in each axis.

5. SEST-INDEX VARIANTS

In spite of the fact that SEST-Index performed well for low change frequencies (between approximately 1% and 13% for storage cost and between 1% and 7% for interval queries and queries based on events), one of its main disadvantages is the rapid growth of its size as the change frequency increases (see Figure 16). This rapid growth is explained by: (1) the greater frequency of snapshots that are needed as the percentage of changes and/or the number of objects increases and (2) each time that an R-tree is created (i.e., a new snapshot), all objects are duplicated including those that have not undergone modification between consecutive snapshots.

In this section we propose two variants of SEST-Index: the first aims to solve the problem of duplication of objects in the snapshots, and the second tries to increase the size of

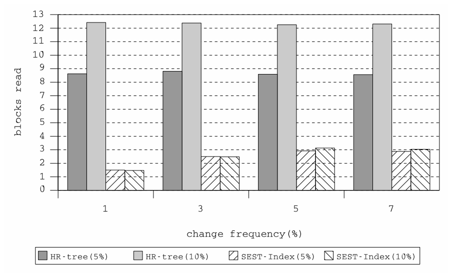

Figure 15: Blocks read by HR-tree and SEST-Index for queries about events for 3000 objects. XX-tree(i%) indicates SEST-Index or HR-tree for queries about events whose query window is formed by i% of the dimension in each axis.

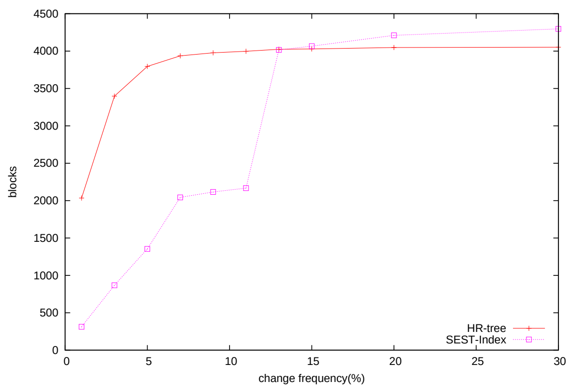

Figure 16: Growth of the index size versus the percentage of change frequency (3000 objects)

the log without affecting the performance of the queries.

5.1 First variant (Partitioning the space)

Our first variant consists in partitioning the original space into a set of disjoint regions. To do so, we use a spatial access method such as K-D-B-tree [9] (we choose this structure because it splits the space into disjoint subspaces while the union of these subspaces comprises the whole space). Then, the logs can be assigned to the regions of the lowest levels of the partition or to regions of intermediate levels (Figure 17). With this idea, duplication of objects can be avoided where no changes have occurred. The index I of Figure 17 is used to rapidly locate the appropriate snapshot for the time given in the queries.

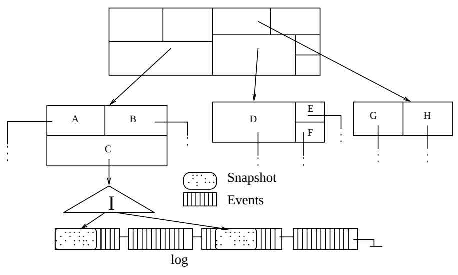

Figure 17: General diagram of the first variant (partitioned with K-D-B-tree)

The following procedure is used to process a *time slice* query. First, regions in the K-D-B-tree that intersect the query window are selected. Then, for each of these regions, and using the index I, the corresponding snapshot is located and the changes are applied to obtain the set of objects in the region that are part of the answer.

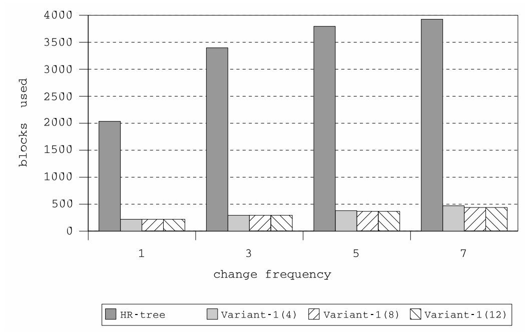

Figure 18: Space usage with 3000 objects (HR-tree versus first variant)

A preliminary experiment compared this first variant of SEST-Index with HR-tree by using a database with 3000 objects, four different change frequencies (i.e., 1, 3, 5 and 7%), and three different log sizes (i.e., 4, 8 and 12 blocks). The results show that this SEST-Index variant requires, approximately, only 10% of the space used by HR-tree (Figure 18), half of the number of block read by HR-tree to process time interval queries (Figure 19), and twice the number of blocks read by HR-tree to process time slice queries (Figure 20), which is similar to, and not much worse than, the

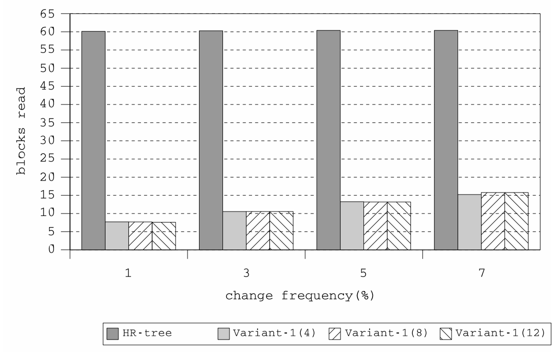

Figure 19: Blocks read in time interval queries with size of time intervals equal to 10 and query windows formed with 10% of the dimension in each axis (HR-tree versus first variant)

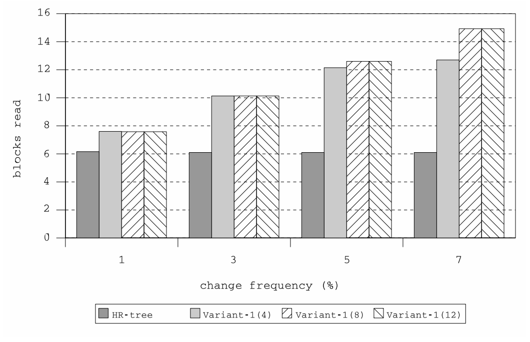

Figure 20: Blocks read in a time slice query with a query window formed by 10% of the dimension in each axis (HR-tree versus first variant)

results obtained with the original SEST-Index.

This approach presents two limitations: (1) the objects must be points and (2) the space or region where the changes occur must be fixed.

5.2 Second variant (Indexing the log)

The second variant is motivated by the results presented in Figures 21 and 22 where it is possible to see that (1) the size of the log has a high impact on the size of the index and (2) a low impact on the performance of the queries. Furthermore, the second variant removes the limitations of the former one, that is, objects may have a geometric extent (i.e., not only points) and the boundaries of the partitions in the space can vary.

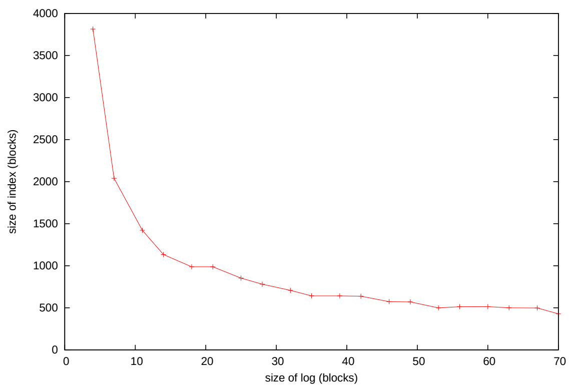

Figure 21: Size of the log versus size of the index (3000 objects, 7% of change frequency)

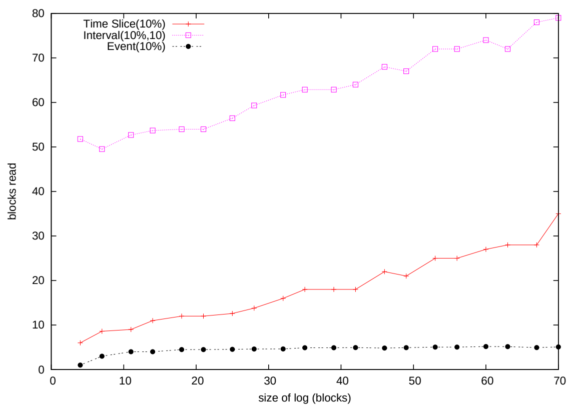

Figure 22: Blocks read for the queries (3000 objects, 7% of change frequency)

This approach consists in maintaining an index for the logs (in addition to the R-tree of the snapshots) for each time instant in which changes occur (see Figure 23).

In Figure 23, R_0 corresponds to an R-tree of the snapshot created at t_0; R_1 and R_2 are the R-trees that index the elements stored in the logs; a, b, c and d are leaf nodes; $b1, b2, b3$ and $b4$ correspond to log nodes and A, B, C and D are internal nodes at the last level of R_0, R_1 and R_2, respectively. The index R_1 is obtained from R_0 in the following

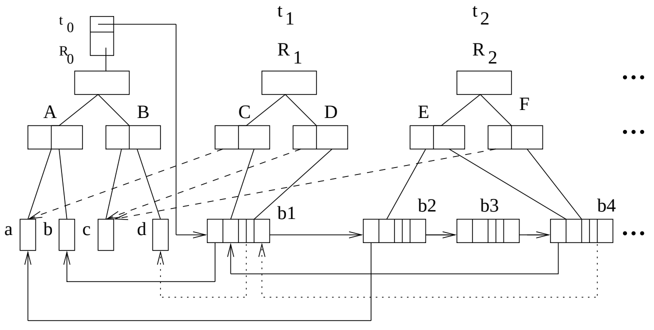

Figure 23: General diagram of the second variant

way: (1) all the internal nodes of R_0 are copied and (2) all the changes that occur within the same MBR of an entry are stored in the same log block together with a pointer to the corresponding leaf node. For example, the changes that occur in the second entry of node C are stored in $b1$ along with a pointer to the leaf node b. The changes cause that the successive MBRs in the insertion path are modified (they grow or decrease) such that they contain the geometry of the objects in the snapshot as well as their changes. The index R_2 can be obtained from R_1 in a similar way.

The number of indexes will depend on the predefined threshold value for some parameter, which can be related to the size of the log or the percentage of overlap of intermediate nodes. Once this threshold value is reached, a new snapshot is created, and the process of the creation of the indexes is repeated. This variant allows us to share log nodes for the indexes; for example, node $b1$ stores changes corresponding to the second entry in C and D. It is also possible that one entry groups many changes and, therefore, such entry requires several blocks. This occurs with the first entry of node E, which stores blocks $b2$ and $b3$ of the log.

Let us suppose that we perform a *time slice* query in time instant t_2 and that the query window intersects only with the MBR of the second entry of F, let e be such entry. We first retrieve the changes that intersect with $e.MBR$ and are in node $b4$, let Q be such set. Afterwards, Q is updated with the changes stored in node $b1$ (indicated by a dotted line). Finally, the set Q is used to update the objects stored in the leaf node d. One of the problems that can come up is that lists of log blocks to be accesed on a query may become too long (for example, dotted line in Figure 23). A way to eliminate this problem is to generate a snapshot at the level of the leaf node in the instant in which the size of the list exceedes a certain pre-established value. This last modification makes this variant be very similar to the former variant, but without the former variant's limitations.

Estimations on the storage used by the second variant indicate that it only requires between 10% and 25% of the space occupied by HR-tree to maintain 3000 objects with a change frequency of 7%. With respect to answer time, the first and second variants have similar performance.

6. CONCLUSIONS

A new method (SEST-Index) is proposed to create spatio-temporal indexes that combine the storage of snapshots and change events over objects. The method tries to establish a compromise between the time required to process the queries and the space occupied by the index.

SEST-Index requires less space than HR-tree, when the change frequency of objects is low (1% to 13%). It also outperforms HR-tree for time interval queries (with change frequency between 1% and 7%) and queries about events. The advantages of SEST-Index over HR-tree in time interval queries increase as the query window increases. For time slice queries, SEST-Index requires more disk accesses than HR-tree, depending on the size of the log. It is important to notice that HR-tree is the most efficient spatio-temporal access method for processing time slice queries known until now.

As further research, we plan to fully implement the two variants of SEST-Index. We will evaluate these variants under different scenarios and with respect to other access methods, such as MV3R-tree and 3D R-tree. We will also define an analytical cost model for our indexes, which will allows us to predict the space usage and response time for query processing.

7. REFERENCES

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

[2] Kollios, G., Tsotras, V. J., Gunopulos, D., Delis, A., and Hadjieleftheriou, M. Indexing animated objects using spatiotemporal access methods. *Knowledge and Data Engineering 13*, 5 (2001), 758–777.

[3] Kollios, G. N. *Indexing Problems in SpatioTemporal Databases.* PhD thesis, Polytechnic University, New York, June 2000.

[4] Manolopoulos, Y., Theodoridis, Y., and J.Tsotras, V. *Advanced Database Indexing*, 1st ed. Kluwer Academic Publishers, 1999.

[5] Mokbel, M. F., Ghanem, T. M., and Aref, W. G. Spatio-temporal access methods. *IEEE Data Engineering Bulletin 26*, 2 (2003), 40–49.

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

[7] 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.

[8] Pfoser, D., and Tryfona, N. Requirements, definitions, and notations for spatiotemporal application environments. In *GIS '98: Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems* (1998), ACM Press, pp. 124–130.

[9] Robinson, J. T. The K-D-B-tree: A search structure for large multidimensional dynamics indexes. In *ACM SIGMOD Conference on Management of Data* (1981), ACM, pp. 10–18.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Tzouramanis, T., Vassilakopoulos, M., and Manolopoulos, Y. Overlapping linear quadtrees and spatio-temporal query processing. *The Computer Journal 43*, 4 (2000), 325–343.

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

[17] 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.