An Access Method for Objects Moving among Fixed Regions

Gilberto Gutiérrez R. [1] Gonzalo Navarro [2] Andrea Rodríguez T. [3]

Resumen *Este artículo resume una propuesta de un método de acceso espacio-temporal para objetos que se mueven a través de un conjunto de regiones fijas y disjuntas. La propuesta se basa en un R-tree, en un método de acceso temporal y en un esquema de hashing. Con nuestro método es posible responder consultas de tipo* time slice, interval *como también consultas sobre la trayectoria que ha seguido un objeto. También es posible procesar objetos espacio-temporales con intervalos de tiempo abiertos, es decir, objetos cuyo tiempo final de permanencia en una determinada posición es desconocido. Esta característica permite mantener un índice espacio-temporal en el cual se pueden mezclar operaciones que modifican la estructura de datos subyacente, con operaciones de consulta.*

Abstract *We propose a spatio-temporal access method for objects that move through a set of fixed and disjoint regions. The proposal is based on an R-tree, on a temporal access method and on a hashing scheme. With our method it is possible to respond to queries of the timeslice and interval types as well as queries about the trajectory that an object has followed. It is also possible to process spatio-temporal queries with open time intervals, that is, queries about objects whose final duration in a determined position is unknown. This characteristic allows maintaining an index in which operations that can modify the underlying data structure can be mixed with query operations.*

1 Introduction

A spatio-temporal database is a spatial database in which objects can change their spatial position and/or their shape in different time intervals. In this way, a spatio-temporal database allows the establishment of models that are very close to the real world, which is essentially dynamic [6]. There exist many applications that require the use of this dynamism; for example, an application whose database needs to store information at all times about the position of all the cars in a fleet of taxis. Other applications can be found in the areas of transportation, environment, social (demographics, health, etc), and multimedia.

For the spatial databases the type of fundamental query is *Window Query* that consists in retrieving all the objects that intersect with a rectangle (window) specified by an user. For the spatio-temporal databases, as well, there exist two types of fundamental queries (which can be considered as subtypes of the *Window Query* type): the first type of query, known as *timeslice* or *timestamp*, allows the retrieving of all the objects that intersect with a rectangle at a specific timestamp. The other type is called *Interval* and allows the retrieving of all the objects that intersect with a rectangle in successive timestamps [13]. The efficient processing of these types of queries (especially *timeslice*) is very important since they are already used as a part of the more complex spatio-temporal queries.

One of the objectives of the Databases Management System is to provide access methods that avoid examining all the objects at the time of the queries. In this way, a Spatio-temporal Databases System must also count on mechanisms that allow the construction of indexes that improve the response time for the spatio-temporal queries. For that purpose, the spatio-temporal access methods that have been proposed are mostly based on three points of view:

[1]Departamento de Auditoría e Informática, Universidad del Bío-Bío, Avenida La Castilla S/N - Chillán(Chile), ggutierr@ubiobio.cl. Partially financed by project "Procesamiento de consultas espacio-temporales", Dirección de Investigación, Universidad del Bío-Bío and RITOS2 (Red Iberoamericana de Tecnologías del Software para la década del 2000).

[2]Departamento de Ciencias de la Computación, Universidad de Chile(Chile), gnavarro@dcc.uchile.cl. Partially funded by Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

[3]Departamento de Ingeniería Informática y Ciencias de la Computación, Universidad de Concepción(Chile), andrea@udec.cl. Partially funded by Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

- *Methods that treat time as another dimension.* For example 3D R-tree proposed in [15].
- *Methods that incorporate the time information* into the nodes of the structure without considering time as another dimension. For example, RT-tree proposed in [16]
- *Methods that use overlapping of the underlying structure of data* reusing that part of the structure in which the stored objects have not suffered changes in their position and/or shape between the consecutive timestamps. For example HR-tree [7, 6] and MR-tree [16].

All the above methods are oriented to support spatio-temporal applications in general domain and therefore do not benefit from the particular characteristics and behavior of the objects of certain types of applications, characteristics which, according to [13, 14], can be convenient to consider for a design of a specific access method. Some of these types of applications are the following:

1. Those in which the objects move at great speed.
2. Applications where the size and the shape of the object is not important and the only point of interest is its position in time [8].
3. Those that require maintaining an on-line index [5]. In this type of index, when an object moves to a new position in the timestamp t_i, its final duration is unknown, its interval being $[t_i, *)$, where $*$ is a variable (always increasing) that represents the current time or now. Subsequently, when the final time, t_j, is known, the interval of duration is updated to $[t_i, t_j)$.

   On the contrary, an off-line index assumes that the position as well as the interval of duration are known. In this way the construction of the off-line index is more efficient since it is possible to rely on more information in an anticipated way [5], however, with an off-line index is not possible to mix operations that update the structure with query operations.
4. Applications in which the space and/or the number of objects does not change.
5. Those where the objects move in previously established areas or sectors; for example applications about road networks and the displacement of vehicles. Furthermore, in this type of applications, it is necessary to query the trajectory that the object has followed.

This paper defines a new spatio-temporal access method that takes into consideration the defined conditions for types 3 and 5 applications. The method allows the processing of queries of the *timeslice* and *interval* type using both historic and actual information about the objects. Furthermore, it is possible to obtain the trajectory of an object and maintain an on-line index. In [3] a similar method of access is proposed, but this one takes into consideration only part of the conditions of type 5 applications since it does not allow the retrieving of the trajectory of an object in an efficient way.

The basic idea of our method is to maintain the intervals of duration of the objects separately for each of the regions or areas as can be seen in Figure 1. For example, in region $R3$ of Figure 2, the intervals of duration of the objects are $[t_1, t_2)$ for the object o_1, $[t_0, t_1)$ for the object o_2 and $[t_5, *)$ for the object o_3. These intervals are indexed using a temporal access method, specifically Window Index described in [10] which is the optimum in time and space for queries of the *pure-timeslice* type. Nevertheless, the basic structure of Window Index as well as its algorithms were modified to allow the processing of spatio-temporal queries and queries based on the trajectories of the objects. The spatial extension of the regions is organized in an R-tree [4] whose basic structure was also modified. In addition to these access methods, an extensible hashing scheme and a skip list are used to allow the retrieving of the trajectory and maintain an on-line index. In this paper only the underlying data structures and the algorithms are described to process the spatio-temporal queries as well as updating the data structures. In a future paper the performance of our access method will be evaluated in relation to other proposals.

The rest of this paper is organized in the following way. Section 2 briefly reviews the access methods, temporal as well as spatial in which our method is based. Section 3 describes the data structures used. Section 4 presents the algorithms used to process the distinct types of queries considered by our spatio-temporal access method. Section 5 shows the algorithm that allows actualizing the data structures. Finally, Section 6 presents the conclusions and future work.

2 Related work.

In this section we review of the temporal and spatial access methods used in our method. The skip

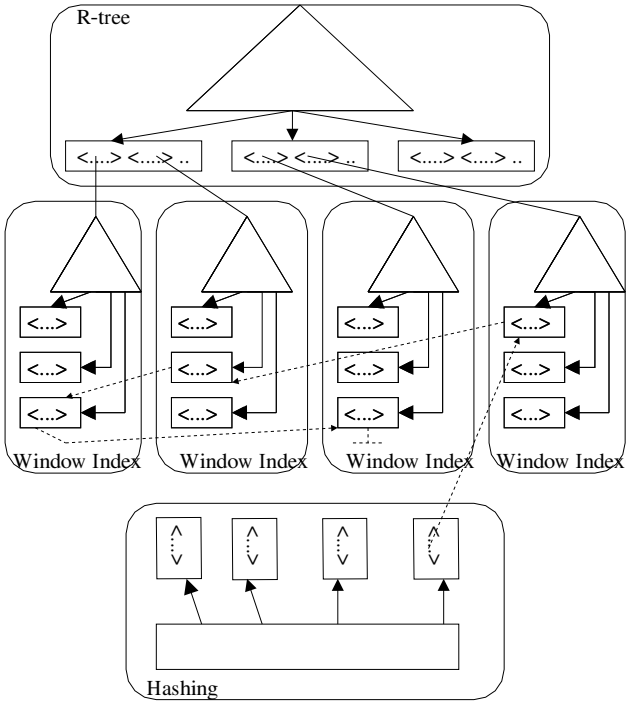

Figure 1: General structure of our spatial-temporal access method

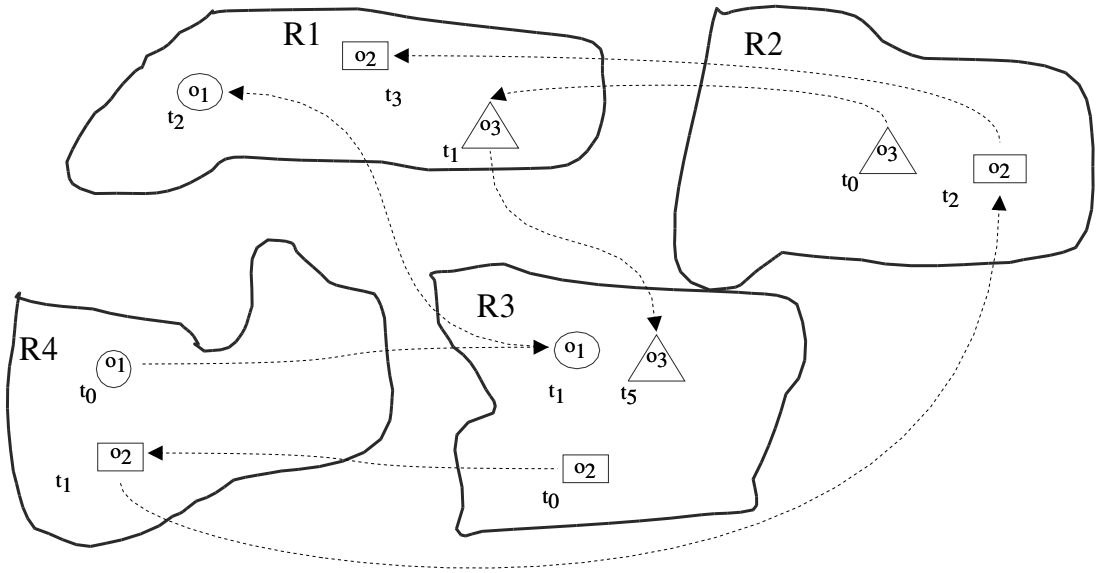

Figure 2: Objects moving around a set of regions

list [9] and extensible hashing [2], used by our proposal too, are not discussed in this section since they are well-known.

2.1 Window Index

Window Index (henceforth WI) [10] is a temporal access method in which each register to be indexed is assumed to have three fields: (1) key attributes that are invariable in time, (2) attributes that change in distinct time intervals where each interval is found to be defined by an initial time and a final time, and (3) a temporal object identifier. Furthermore, it is assumed that:

- When an object is added to the database in the timestamp t_1, its time interval is $[t_1, *]$, where $*$ refers to current time. In this way an object is activated in the timestamp t_1 and will exist until it is eliminated.
- When a previously inserted object is eliminated in the timestamp t_2, its time interval that was originally $[t_1, *]$, is now $[t_1, t_2]$.
- No other operations are defined on the stored objects in the database. For example, it is prohibited to modify the limits of the intervals that are already found established or add an object that refers to a time in the past.

WI is a method for the problem of the intersection of segments, that is, given a set of intervals $[a_i, b_i], 1 \leq i \leq n$, and a point q, to find all the intervals or segments that intersect with q. This problem can be solved easily by identifying, for each a_i and b_i, the set of intervals that intersect and insert them into a B^+-tree. This solution, nevertheless, needs space $O(n^2)$ in the worst case. WI is based on the observation that the adjacent sets do not differ in an important way. In this way, making an adequate choice of the sets, it is possible to reduce the space from $O(\frac{n^2}{B})$ to $O(\frac{n}{B})$, where B is the block size. WI, first orders the extreme points of the input intervals, $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ obtaining $e_1, e_2, \ldots, e_{2n}$. Then a set of windows $W_1, W_2, \ldots, Wp$ is chosen on the extreme points $w_1, w_2, \ldots, w_{p+1}$ such that $w_1 = e_1, w_{p+1} = e_{2n}, w_j \leq w_{j+1}$ and $W_j = [w_j, w_{j+1}]$. For example, in Figure 3 the limits of the window W_2 are a_1 and d_1. In this way the windows represent a set of p partitions of the intervals between e_1 and e_{2n}. For each window W_j there exists a list of intervals that corresponds to those that intersect with the limits of W_j; said list is stored as a B^+-tree indexed by the initial point of the window. WI forces the fulfillment of the following two conditions about the windows to guarantee efficiency in space as well as in time.

1. For each point q that intersects a window W_j, the set of intervals that intersects q (O_q) satisfies the condition $0 < |W_j| \leq \delta\ max(1, |O_q|)$, where $\delta > 1$ is a parameter that provides a tradeoff between space and response time, $|W_j|$ represents the number of intervals in the W_j window and $|O_q|$ is the size of the set O_q.
2. $\sum_{j=1}^{j=p} |W_j| < \frac{2\delta}{\delta - 1} n$ guarantees that the list of intervals of each window needs linear storage.

Making small modifications on the basic scheme of WI, it is possible to use it as an efficient temporal access method. In the temporal model each object is associated with three attributes: (1) a key attribute that is invariable over time, (2) a time interval, and (3) an object identifier (Oid). To index these objects, first a list of intervals is constructed that will now contain temporal objects instead of intervals. Each object in the list of intervals is inserted separately in the B^+-tree considering the compound key *(initial point of the window, attributed key)* as the key to the search. For example, if the objects of the window are considered to be W_1 (Figure 3), the pairs that must be inserted in the B^+-tree are the following: $(a_1, k_1, a), (a_1, k_2, b)$ and (a_1, k_3, c), where (a_1, k_1), (a_1, k_2) and (a_1, k_3) are the key compounds of the objects a, b, and c respectively.

WI also allows indexing intervals whose extreme end is unknown (dynamic version of the problem). Finally, WI is optimum in space ($O(\frac{n}{B})$) and time ($O(log_B n + \frac{t}{B})$, for queries of the *pure-timeslice* type, where n is the number of changes, B is the capacity of the block and t is the size of the response.

2.2 R-tree

A R-tree is an extension of B-tree for multidimensional objects (points and regions) [4, 12]. Each node corresponds to a page or block of disk. The leaf nodes of an R-tree contain entries of the form (I, Oid) where Oid is the identifier of the spatial object in the database and I is an n-dimensional rectangle that corresponds to the Minimum Bounding Rectangle(MBR) of the spatial object, that is, $I = (I_0, I_1, \ldots, I_n)$, where n is the number of dimensions and I_i is a closed interval $[a, b]$ that describes the limits of the object in the i dimension. The internal nodes (non-leaf nodes) contain entries of the

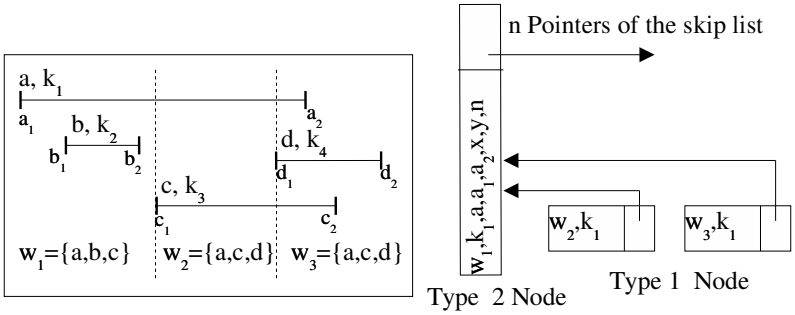

Figure 3: Types of entries considered by a Window Index

form $(I, pchild)$ where *pchild* is the direction of the corresponding leaf node in the R-tree and I covers all the rectangles defined in the entries of the child node.

M being the maximum number of entries that can be stored in a node and being $m \leq \frac{M}{2}$ a parameter specifying the minimum number of entries in a node. An R-tree satisfies the following properties [4]:

1. Each node contains between m and M entries unless it corresponds to the root.
2. For each entry (I, oid) in a leaf node, I is the minimum rectangle that (spatially) contains the object.
3. Each internal node has between m and M children, unless it be the root.
4. For each entries of the form $(I, pchild)$ of an internal node, I is the smallest rectangle that spatially covers the rectangles defined in the child node.
5. The root node has at least two children, unless it is a leaf.
6. All the leaves are found at the same level.

The height of the R-tree that stores N keys is at least $\lceil log_m N \rceil - 1$, since the number of children is at least m. The maximum number of nodes is $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{m^2} \rceil + \cdots + 1$ [4]. The use of storage (worst case) for all the nodes, except the root, is $\frac{m}{M}$ [4]. The nodes have a tendency to maintain more than m entries that allows the height of the tree to decrease and improve storage use.

The search and insertion algorithms are very similar to the algorithms used for these operations in a B-tree.

Many variants exist of the R-tree; one of these is proposed in [11] called R^+-tree, which tries to avoid that in a search various paths of the tree cross each other, that is achieved by storing the MBR of an object in more than one page when it intersects with various internal nodes. The other proposal is the one presented in [1] known as R^*-tree, which introduces a policy of insertion called forced reinsertion that consists in not dividing a node immediately when it fills up. Instead of this, it proposes eliminating p entries of the node and reinserting them in the tree. Furthermore, the algorithms minimize the superposition of regions, the MBR perimeters, and maximize storage use.

3 Our spatio-temporal access method

As we have already said, our spatio-temporal access method focuses on supporting applications in which there exist a set of fixed, disjointed, and preestablished areas or regions around which the objects move (Figure 2). We assumed that the objects are capable of informing, in a discrete way, the coordinates and the time in which an object reached a new spatial position that must belong to one of the regions. We defined two types of object displacement: (1) *interregions* when the object is displaced to a position that belongs to a region that is distinct to the one where it is actually found, and (2) *intraregions* when the object is moved to a position within the same region.

When an object is displaced to a position p in the timestamp t_1, its interval of duration in p is $[t_1, *)$. Subsequently when the object is displaced

to a position q in the timestamp t_2, the time interval that the object remains in p is $[t_1, t_2)$ and the interval of duration in q is $[t_2, *)$. In this way an object is found in a position p in any timestamp t so that $t_1 \leq t < t_2$.

To maintain the spatial and temporal information of the objects, our proposal considers a spatial (R-tree) access method and a temporal (Window Index) access method. R-tree is used to construct a spatial index around the regions or areas. As far as WI is concerned (one per area or region), it is used to index the time intervals in which the objects have remained in the corresponding region, and is used to carry out searches based on time (see Figure 1). In this way, a query of the *timeslice* type, for example, is solved by first retrieving the areas or regions that intersect within the range or spatial window of the query and then, using the respective WI, the objects whose intervals of duration in the region intersect with the time given in the query are retrieved. Our method also considers a hashing scheme organized by the *Oid* do the object and has the purpose of retrieving the trajectory of an object and allows to maintain open intervals of duration that which makes possible an on-line index. The trajectory of the objects is maintained through an index of skip lists (one for each object) and corresponds to an entry type in the WI's.

3.1 Data structures of our proposal

In this section, the data structures and the algorithms that make up our spatio-temporal access method are discussed.

3.1.1 R-tree

As above-mentioned, the R-tree will store the spatial component of the problem, that is, the regions or areas. The structure of the R-tree nodes is the following:

Internal Node $< MBR, ptr >$ same as the data structure of the original proposal [4].

Leaf Node $< MBR, ptrArea, ptrWI >$, where MBR is the minimum rectangle that surrounds the extension of the area, $ptrArea$ is a block pointer that stores the points defined by the area (together with other attributes), and $ptrWI$ is the block pointer that allows the attainment of the area's WI.

3.1.2 Window Index

This structure is used to store the time intervals in which the objects remain within the area and/or spatial position. It is based principally in a B^+-tree that is organized by a compound key formed by the start of each window and an object key. WI can duplicate objects in various windows with the purpose of guaranteeing optimum response times. Given that in our proposal these entries are part of the skip list of the spatio-temporal object, we only considered the first entry (first window) as node of the skip list. The remaining entries (duplicated) maintain a pointer to the first as is shown in Figure 3. In Figure 3, WI stores the object a in three different windows (W_1, W_2, W_3), but in our proposal, only in the W_1 window we store the data of the time interval, position, *Oid*, and the pointers of the skip list of the trajectory of the object *Oid*. There are two types of entries of a WI block of data:

1. $< te, w, k, pe >$ (see Type 1 Nodes in Figure 3), where te indicates the entry type, w is the window number, and k is the object key. Finally, pe is the block pointer in which the first entry of the object k is stored and that corresponds to the node of the skip list of the trajectory of the object *Oid*.

2. $< te, w, k, Oid, t_e, t_s, x, y, n, < p_0, p_1, \ldots, p_{n-1} >>$ (see Node Type 2 in Figure 3). Where te, w, k and *Oid* represent the same as in the above-mentioned entry. te corresponds to the time in which the object is moved to the position (x, y). In the same way t_s corresponds to the timestamp in which it left such position. n corresponds to the number of pointers or node level in the skip list. p_0, p_1, p_{np-1} correspond to the node pointers of the level 0, 1, etc of the skip list.

The entries in the internal blocks have the following format: $< ptr, (w, k) >$, where w is the window number, k the object key, and ptr a block data pointer and/or internal block whose stored keys are greater or equal to the compound key (w, k).

3.1.3 Hashing scheme

We used a hashing scheme (extensible hashing) to maintain and index for the *Oid* of the objects. This allowed us to do searches based on the objects (actual location, trajectory, etc) as we will see later on. An entry (register) in the hashing scheme has the following structure: $< Oid, p_0, p_1, ..p_{nm-1} >$

where *Oid* corresponds to the object identifier, p_0, p_1, p_{nm-1} correspond to node pointers of the object skip list. nm is a parameter that indicates the maximum level of the nodes of the skip list.

4 Query processing

4.1 *Timeslice* Queries

This query consists in retrieving all the objects that are found in a determined sub-space R in a timestamp t. Figure 4 shows the algorithm to process this type of query. The *SearchInRtree(R)* function corresponds to the search algorithm in an R-tree proposed in [4]. The *SearchInWindowIndex(e.ptrWI, t)* function is used to find the objects that have remained in R in the timestamp t.

4.2 *Interval* Queries

This type of queries is very similar to the *timeslice* type. The difference resides in that the temporal component considers a time interval instead of a timestamp. In this way a query of this type consists in retrieving all the objects that remain in a sub-space R during the time interval $[t_i, t_j]$. Basically, this type of query can be solved with the same algorithm found in Figure 4, only being necessary to replace the *SearchInWindowIndex(e.ptrWI, t, t)* instruction by *SearchInWindowIndex(e.ptrWI, t_i, t_j)*.

4.3 Retrieving of the trajectory of an object

In general, we can define the trajectory of an object as the set of spatial positions in which an object remains during a determined time. In our proposal it is possible to obtain two types of trajectories. The first type consists in obtaining all the areas or regions in which an object has remained. For example, in Figure 2 for the object o_2, the trajectory is $< R3, R4, R2, R1 >$. The second type consists in retrieving the coordinates where a determined object has remained in a period of time. The algorithm in Figure 5 allows obtaining the answer to the queries belonging to the second type.

5 Update of the data structure

In our model, when an object is displaced from one position to another, it is necessary to update the underlying structure of data. Furthermore, this is the only event that provokes an actualization. The same as in the temporal access method WI, an object is displaced to a position p in a timestamp t_i, the interval of duration of the object in the position p will be $[t_i, *)$. Subsequently, when the object is displaced to a position q that is distinct from p in the timestamp t_j, with $t_j > t_i$, the interval of duration of the object in p will be $[t_i, t_j)$ and in q will correspond to $[t_j, *)$. Figure 6 describes the algorithm that allows the actualizing of the data structure.

```
// Given a rectangle R and a time t, find all the objects that intersect with R in the time t.
SetOid TimeSliceQuery(R, t) }
  Being E the set of entries of the leaf nodes of the R-tree whose MBR intersect with R.
  Being Q the set of objects that satisfy the query
  Q = ∅
  E = SearchInRtree(R)
  For each of the entries in E, its corresponding WI is revised, retrieving all the objects
  that meet the spatial and temporal restrictions given in the query
  for each(e ∈ E)
       Q = Q ∪ SearchInWindowIndex(e.ptrWI, t, t)
  return Q
}
```

Figure 4: Algorithm to process a query of the type *timeslice*

```
// Given an object Oid and a time interval, [t_i, t_j], obtain the trajectory
// of the object.  It is assumed that t_i ≤ t_j.
Trajectory Positions(Oid,t_i, t_j) {
// T is a set of spatial positions.  T is used to maintain the trajectory
  T = ∅
  // Search in the hashing scheme the corresponding entry to the object identified
  // as Oid.  Being e this entrie.
  e = SearchInHashing(Oid)
  //With the entry e, search the node of the skip list whose final time (tmp.t_s)
  //is the least time so that t_j ≤ tmp.t_s.  The node being tmp.
  tmp = SearchInSkipList(e.p_0, e.p_1, ..., e.p_{n-1}, t_i)
  // go through the nodes of the skip list starting with tmp
  while(nodes are left and tmp.t_e ≤ t_i ) {
  // an element is added to the answer set T.  Each element is formed by the position (x.y)
  // and the time interval in which the object was in said position.
     T = T∪ < tmp.t_s, tmp.t_e, tmp.x, tmp.y >
     tmp = tmp.p_0
     }
  return T
}
```

Figure 5: Algorithm to obtain the trajectory of an object

```
// This algorithm allows updating the data structure when an object Oid is
// displaced from a position to a new one (x, y) in the timestamp t.
Move(Oid, t, x, y) {
   // Search in the hashing scheme using as the search key for Oid an
   // entry e with values < Oid, p_0, p_1, ..p_{n-1} >.  Being p the direction of the
   // block that contains the entry e.
   p = SearchInHashing(Oid)
   e = SearchInBlock(p, Oid)
   // tmp maintains the first node of the skip list
   tmp = e.p_0
   // Search the region or area where the object is actually found.  Being s this entrie.
   s = SearchInRtree(tmp.x, tmp.y)
   // Update the entry in WI s.PtrWI with the values < tmp.t_e, t >.
   UpdateWindowIndex(s.PtrWI, tmp.t_e, t)
   // Search the region or area of the object's displacement according to the coordinates
   // (x, y) and using the R-tree.  Being r this entrie.
   r = SearchInRtree(x, y)
   // Being n the level of the node inserted in the skip list
   n = NodeLevel()
   Being ne an entry < Oid, t_e, x, y, n, < p_0, p_1, ..., pn - 1 >>
   for(i = 0; i < n; i++)ne.p_i = e.p_i
   bid = InsertinWindowIndex(r.PtrWI, ne)
   for(i = 0; i < n; i++)e.p_i = bid;
   Update the block p with the new values of the entry e.
}
```

Figure 6: Algorithm to update the data structure

6 Conclusions

Our spatio-temporal access method allows the processing of queries of the *timeslice* and *interval* types for objects that move around a set of regions with pre-established and disjoint boundaries between themselves. An advantage of our method with respect to an FNR-tree, proposed in [3], is that it is possible to process queries around the trajectory that the objects have followed and furthermore open time intervals can be managed which is achieved by modifying the basic Window Index structure. This last advantage allows maintaining a spatio-temporal on-line index, and it is therefore possible to mix query operations with operations that update the index. Finally, all the data structures on which our access method is based are quite known (except Window Index), so it is presumed that the implementation is relatively simple. In the future, we will try to implement and evaluate our proposal comparing it with the best methods proposed in the literature.

References

[1] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The R*-Tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD Conference on Management of Data* (1990), ACM.

[2] Fagin, R., Nievergelt, J., and Pippenger, N. Extendible hashing-a fast access method for dynamic files. *ACM Transactions on Database Systems 4*, 3 (1979), 315–344.

[3] Frentzos, E. Indexing objects moving on fixed networks. In *Advances in Spatial and Temporal Databases* (2003), pp. 289–305.

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

[5] 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.

[6] Nascimento, M., Silva, J., and Theodoridis, Y. Access structures for moving points, 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., Jensen, C. S., and Theodoridis, Y. Novel approaches in query processing for moving object trajectories. In *The VLDB Journal* (2000), pp. 395–406.

[9] Pugh, W. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures* (1989), pp. 437–449.

[10] Ramaswamy, S. Efficient indexing for constraint and temporal databases. In *ICDT'97 Intern. Conference on Database Theory, Delphi, Greece* (1997).

[11] Sellis, T., Roussopoulos, N., and Faloutsos, C. The R^+-Tree: A dynamic index for multi-dimensional objects. In *13th Conference on Very Large Data Bases* (Brighton, England, 1987), pp. 507–518.

[12] Sellis, T., Roussopoulos, N., and Faloutsos, C. Multidimensional access methods. Trees Have Grown Everywhere. In *23rd Conference on Very Large Data Bases* (Athens, Greece, 1997), pp. 13–14.

[13] Tao, Y., and Papadias, D. Efficient historical R-Tree. In *IEEE International Conference on Scientific and Statical Database Management* (2001).

[14] 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.

[15] Theodoridis, Y., Vazirgiannis, M., and Sellis, T. K. Spatio-temporal indexing for large multimedia applications. In *International Conference on Multimedia Computing and Systems* (1996), pp. 441–448.

[16] 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.