Dynamic Queries over Mobile Objects*

Iosif Lazaridis[1], Kriengkrai Porkaew[2], and Sharad Mehrotra[1]

[1] University of California, Irvine, USA
[2] King Mongkut's University of Technology Thonburi, Bangkok, Thailand

Abstract. Increasingly applications require the storage and retrieval of spatio-temporal information in a database management system. A type of such information is *mobile objects*, i.e., objects whose location changes continuously with time. Various techniques have been proposed to address problems of incorporating such objects in databases. In this paper, we introduce new query processing techniques for *dynamic queries* over mobile objects, i.e., queries that are themselves continuously changing with time. Dynamic queries are natural in situational awareness systems when an observer is navigating through space. All objects visible by the observer must be retrieved and presented to her at very high rates, to ensure a high-quality visualization. We show how our proposed techniques offer a great performance improvement over a traditional approach of multiple instantaneous queries.

1 Introduction

Many new applications require storage and retrieval of spatio-temporal data. A class of such data is *mobile objects*, i.e., objects whose location changes continuously with time. Mobile objects require special handling from the database system, since unlike most other item types, their attributes change continuously. Thus, using traditional approaches, very high update rates must be sustained by the DBMS for complete accuracy of the objects' representation.

Our motivating application is that of a large scale situational awareness system in which a large volume of spatio-temporal information must be accessed and presented to the user at interactive rates as she navigates through virtual space: all objects within her view frustum must be retrieved and presented. A reasonable approach is to use a database system for storage and a spatial index for data access. The rate at which queries are posed is very high. If retrieval performance is weak then either (i) the frame rate will drop ("choppy" motion), or (ii) geometry will be omitted, (inaccurate presentation).

In this paper we explore a special kind of query, *Dynamic Query* (DQ) arising in spatial databases. A DQ, associated with a mobile observer, is evaluated

* This work was supported in part by the National Science Foundation under Grant No. IIS-0086124, in part by the Army Research Laboratory under Cooperative Agreements No. DAAL-01-96-2-0003 and No. DAAD-19-00-1-0188, and in part by Air Force Grant F33615-01-C-1902. We thank the anonymous reviewers for their detailed and helpful comments.

continuously as the observer moves in space. In nearby locations many of the objects retrieved by the query will tend to be the same. It makes sense to conserve disk I/O by retrieving them only once, reusing them for as long as they are visible. This paper provides techniques to achieve this for both *Non-Predictive Dynamic Query* (NPDQ) where the observer's motion is not a priori known and in *Predictive Dynamic Query* (PDQ) where it is known.

The notion of a Dynamic Query applies equally well to static objects coexisting in an application with mobile ones. Consider a query being generated by a vehicle during a military exercise. This may involve friendly and enemy vehicles, also landmarks (e.g., natural obstructions), temperature readings from field sensors, mine fields, etc., which are static. Our query evaluation algorithms can be used for static objects as well, since these are a special case of mobile ones.

The rest of the paper is organized as follows: in Sect. 2 we review related work for indexing and retrieving mobile objects. In Sect. 3 we describe how we can represent and index motion. In Sect. 4 we deal with dynamic queries and their implementation. Specifically, in Sect. 4.1 we deal with PDQ, in which the trajectory of the observer is known beforehand while Sect. 4.2 with the general case (NPDQ, unknown trajectory). In Sect. 5 we present some results from the experimental evaluation of our techniques, supporting their effectiveness. In Sect. 6 we conclude, presenting directions of future research.

2 Related Work

The problem of indexing and querying mobile objects in database systems has been studied in the literature [25, 8, 19, 14, 15, 18, 24]. Most of this work uses a multidimensional index structure (R-Tree and its family [5, 2, 22], Quadtree [21], or hB-tree [10]). For temporal objects special indexing techniques have been proposed such as multi-version index structures, e.g., Time-Split B-tree (TSB-tree) [11], multi-version B-tree [1], and others, summarized in [20]. Proposals for spatio-temporal index structures have been summarized in [27].

All these data structures focus on static objects whose value changes explicitly with an update. Recent work has explored indexing for dynamic properties (which may change without explicit update). Research has focused on indexing and query processing over mobile data [25, 8, 19, 14, 15]. Most of this work deals with spatio-temporal range queries. [24] deals with nearest-neighbor queries.

Our work in [14, 15] classifies selection queries, including spatio-temporal range and nearest neighbor queries on both temporal and spatial dimensions. Algorithms for these types of queries are presented using Native Space Indexing (NSI) in which indexing is performed in the original space where motion occurs and Parametric Space Indexing (PSI) where a space defined by motion parameters is used. A comparative study between the two indicates that NSI outperforms PSI, because of the loss of locality associated with PSI. In the present, we use NSI exclusively; dynamic queries can be applied to PSI as well.

3 Motion Representation and Indexing

We now describe how we represent and index the motion of objects. The following forms a background for the query processing schemes to be discussed in Sect. 4.

3.1 Motion Representation

Objects are translating continuously in a d-dimensional space. In most spatial applications, d is 2 or 3. We ignore other motion types (e.g., rotation), since an object's visibility is dictated by its position. The location vector $\bar{x} = (x_1, ..., x_d)$ of object O changes with time, so we write $O.\bar{x} = f(t, \bar{\theta})$, where t is time and $\bar{\theta}$ is a parameter vector (e.g., initial location, speed, etc.).

We observe that the object's motion changes continuously making it impossible to maintain its precise location at every instance of time. This would entail a very high number of updates being generated. Instead, we use the following model. The object (or sensors tracking it), sends at time $O.t_l$ an update of its motion information. This consists of a time interval $O.\bar{t} = [O.t_l, O.t_h]$ in which this update is valid and a vector of *motion parameters* $O.\bar{\theta}$. The database can then, for all queries involving time $t \in O.\bar{t}$ deduce the object's location as $O.\bar{x} = f(t, \bar{\theta})$, using *location function* f which returns the location of an object at time t given its parameters $\bar{\theta}$ stored in the database.

Consider an object translating linearly with constant velocity. At each update the parameter vector is $\bar{\theta} = (O.\bar{x}_{t_l}, O.\bar{v})$, i.e., the object's initial location $O.\bar{x}_{t_l}$ at time $O.t_l$ and its constant vector velocity $O.\bar{v}$. Subsequently, we can easily write the object's location function as:

$$O.\bar{x} = f(t, O.\bar{x}_{t_l}, O.\bar{v}) = O.\bar{x}_{t_l} + O.\bar{v} \cdot (t - O.t_l), \forall t \in O.\bar{t} = [O.t_l, O.t_h] \quad (1)$$

If at some time t' velocity $O.\bar{v}$ changes, then the information stored in the database becomes imprecise ($f(t', \bar{\theta}) \neq O.\bar{x}$) unless an update is issued immediately. Such a change is expected to occur frequently (even continuously); it is thus unreasonable to issue an update whenever needed. Instead, we only issue an update if the object's location (as deduced by the database, by applying f, given $\bar{\theta}$) differs from its current one by more than a threshold value. Thus, the error in the database representation of each object is bounded.

There is a natural tradeoff between the cost of an update (which depends on the update frequency) and the precision of information captured by the database. The issues involved in update management (including this tradeoff) were dealt with in [28]. If we take into account the uncertainty about the location of an object, then its position at any instance of time will no longer be a point (as per the above given location function) but rather it will be a bounded region that captures the *potential* location of the object. More generally, we can think of a distribution $f(\bar{x}, t, \bar{\theta})$ capturing the probability that an object will be in some location $\bar{x}$ at time t, given its last update $\bar{\theta}$. A thorough treatment of the subject of managing the uncertainty of object locations can be found in [12].

For simplicity of exposition, we will assume that the object's location can be precisely determined via f. We can easily generalize to the imprecise case. As we

will see, a motion segment's bounding rectangle (minimal rectangle containing its trajectory between updates) is used for indexing. If the object's location is imprecise, then a larger bounding rectangle for the motion will be used, resulting in some false admissions. This is to be expected: allowing for imprecision entails retrieving objects that in reality do not fall within the query region. However, no objects will be missed, as the larger "imprecise" bounding box always contains the smaller "true" bounding box of the motion.

We will be using the following notation for the rest of the paper:

Definition 1 (Interval). *$\overline{I} = [l, h]$ is a range of values from l to h. If $l > h$, I is an empty interval ($\varnothing$). A single value v is equivalent to $[v, v]$. Operations on intervals that we will use are intersection ($\cap$), coverage ($\uplus$), overlap ($\between$) and precedes ($\preceq$). Let $\overline{J} = [J_l, J_h]$ and $\overline{K} = [K_l, K_h]$.*

$$\overline{J} \cap \overline{K} = [\max(J_l, K_l), \min(J_h, K_h)] \qquad \overline{J} \between \overline{K} \Leftrightarrow \overline{J} \cap \overline{K} \neq \varnothing$$
$$\overline{J} \uplus \overline{K} = [\min(J_l, K_l), \max(J_h, K_h)] \qquad \overline{I} \preceq \overline{J} \Leftrightarrow \forall P \in \overline{I} : P \leq J_l$$

Definition 2 (Box). *$\square B = \langle \overline{I}_1, \overline{I}_2, \ldots, \overline{I}_n \rangle = \langle \overline{I}_{1\ldots n} \rangle$ is an n-dimensional box covering the region of space $\overline{I}_1 \times \overline{I}_2 \times \ldots \times \overline{I}_n \subset \Re^n$. A box $\square B$ may be empty ($\square B = \varnothing$) iff $\exists i : \overline{I}_i = \varnothing$). An n-dimensional point $p = \langle v_{1\ldots n} \rangle$ is equivalent to box $\langle [v_1, v_1], [v_2, v_2], \ldots, [v_n, v_n] \rangle$. By $\square B.\overline{I}_i$ we note a box's extent along the i^{th} dimension. Operations on boxes are the same as those on intervals.*

3.2 Motion Indexing

We summarized motion indexing work in Sect. 2. Here, we describe the Native Space Indexing (NSI) technique of [14, 15], following the framework proposed in [27]. This was shown [15] to be effective for indexing mobile objects and is thus used in this paper, in conjunction with the new Dynamic Query algorithms.

Consider an object O translating in space. Its motion is captured by a location function $O.\bar{x}(t)$ [1]. Let the valid time for the motion be $O.\bar{t}$. The motion is represented as a bounding rectangle with extents $[min_{t \in O.\bar{t}} x_i(t), max_{t \in O.\bar{t}} x_i(t)]$ along each spatial dimension i, and an extent $O.\bar{t}$ along the temporal dimension. A multi-dimensional data structure (e.g., R-Tree) is then used to index this Bounding Box (BB) representation of the motion. The index will contain multiple (non-overlapping) BBs per object, one per each of its motion updates.

We will now show how spatio-temporal selection range queries can be answered using this index. The query is given as a rectangle Q with extents along the spatial and temporal dimensions. Intuitively, this means: "retrieve all objects that were in a rectangle of space, within an interval of time". Evaluating the query is straightforward using the normal index range search algorithm. As

[1] Since we will be assuming precision of information, we can assume that the object's location $O.\bar{x}$ is precisely given by the location function f. If we omit the parameter vector $\bar{\theta}$ this assumes that it is understood that the location function is applied with the appropriate $\bar{\theta}$ for the time on which the query is issued.

an example, for an R-Tree, all children nodes of the root whose BBs intersect with Q will be retrieved, and subsequently all children of these with the same property, and so on, all the way to the leaf level in which the actual motion segment BBs are stored. More formally, an R-Tree node with bounding rectangle R will be visited, given a query Q iff $R \between Q$.

An optimization introduced in [13] and [14, 15] follows from the observation that a motion's BB may intersect with Q, while the motion may not. Since the motion is represented as a simple line segment, it is simple to test its intersection with Q directly. Thus, at the leaf level of the index structure, actual motion segments are represented via their end points, not their BBs. This saves a great deal of I/O as we no longer have to retrieve motion segments that don't intersect with the query, even though their BBs do. The precise way of checking the intersection of a query rectangle with a line segment is given in [14, 15].

Spatio-temporal range queries are only one of the possible types of interesting queries in an application with mobile objects. Other types of selection queries, e.g., nearest neighbor search in either the temporal or spatial dimensions were studied in [14, 15]. Other query types, e.g., distance joins [6] are also possible. In the present we will deal with spatio-temporal range queries.

4 Dynamic Queries and Associated Algorithms

The notion of a *Dynamic Query* over mobile objects was introduced in the classification of [23] where it was named Continuous Query. Such a query changes continuously with time, as contrasted with a a *Snapshot Query* [14] which is posed to the system once, is evaluated and then ceases to exist (and can thus be thought of as taking a snapshot over the database at some particular time). A Dynamic Query has a different meaning from continuous queries discussed in [4, 26], and from triggers (also evaluated continuously), in that the condition ("query window") changes with time.

Definition 3 (Snaphost Query). *A Snapshot Query Q is a selection query for all motion segments intersecting box $\langle \bar{t}, \overline{x}_1, \ldots, \overline{x}_d, \rangle$ in space-time. d is the space dimensionality (usually 2 or 3).*

Note that we define a snapshot as having temporal extent. In visualization we need the special case in which this is reduced to a single time instance.

Definition 4 (Dynamic Query). *A Dynamic Query DQ is a series of snapshot queries $Q_1, Q_2, \ldots, Q_n$ such that $Q_i.\bar{t} \preceq Q_{i+1}.\bar{t}, i = 1, \ldots, n-1$.*

Dynamic queries arise naturally in spatio-temporal environments where the observer and/or the objects are in continuous motion. For example, a mobile object may wish to continuously monitor other objects in its vicinity – for instance, within a distance δ of its location at each instance of time. Similarly, in a virtual environment, in order to support a fly-through navigation over a terrain, objects in the observer's view frustum along her trajectory need to be continuously retrieved. A fly-through over the terrain can be thought of as a dynamic

query for which each frame rendered corresponds to a snapshot range query for objects in the observer's view at that time. For smooth motion perception, the renderer may pose to the database 15-30 snapshot queries/sec.

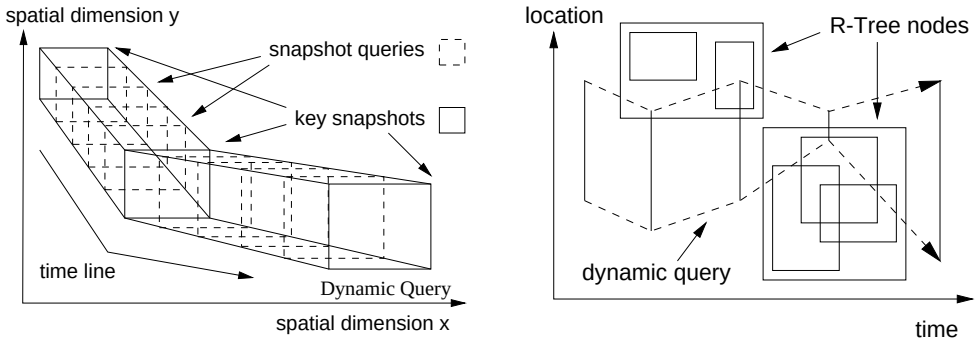
A naive approach to handling dynamic queries is to evaluate each snapshot query in the sequence independently of all others. Doing so is far from optimal since the spatial overlap between consecutive snapshot queries can be very high. Imagine a user moving in a virtual environment at a speed of 500 km/hour and her view at each instance of time is a 10 km by 10 km window in the terrain. Given that 20-30 snapshot queries/sec are fired at the database (one per frame of the visualization), the spatial overlap between consecutive snapshots comes out to be approximately 99.9%. Even if the objects being visualized are themselves mobile, a great deal of overlap is expected in the results of successive queries. It makes sense to exploit this overlap, avoiding multiple retrievals of objects that are relevant for more than one snapshot queries. A natural question is to ask whether or not handling this overlap explicitly is beneficial or not, since objects retrieved in the previous query will already be buffered in main memory, e.g., if we use an LRU buffer. Unfortunately, this is not the case, since buffering takes place at the client (where the results are rendered), and not at the server (where retrieval is done). If each session (e.g., a fly-through by some user) used a buffer on the server, then the server's ability to handle multiple sessions would be diminished. More importantly, there would be significant communication overhead in transmitting large volumes of geometrical data from the server to the client.

Recall that we classify dynamic queries as predictive (PDQ) and non-predictive (NPDQ). In the first case, the sequence Q_i is known a priori and can be used to create an I/O optimal algorithm for them. In the non-predictive (more general) case, knowledge of Q_i allows us to make no inferences about Q_{i+1}, i.e., the query changes arbitrarily.

Both types of queries occur frequently in large-scale visualization. PDQ corresponds to "tour mode" in which the user follows a pre-specified trajectory in the virtual world. It is also useful in situations where the user's trajectory changes infrequently; this is common, since usually the user changes her motion parameters in the virtual world every few seconds by appropriate interaction. In the interim – corresponding to hundreds of frames – her motion is predictable based on her last motion parameters. Algorithms for NPDQ are also very useful, because it is precisely at the times of maximum user interaction, when the user's view frustum changes abruptly that the database subsystem is pushed to its limit, trying to load up new data from disk for the changing query parameters. Our algorithms for NPDQ help alleviate this problem.

A system using the concept of Dynamic Queries would operate in three modes:

• **Snapshot**.— The query changes so fast that there is little overlap between Q_i and Q_{i+1}. Thus, multiple snapshot queries are used. This case is very rare in visualization, in which motion is smooth. In this context it would correspond to Q_i and Q_{i+1} being completely different; this might happen e.g., if the user is "teleported" from one part of the virtual world to another.



(a) Dynamic Range Query in Spatial Dimensions

(b) Query and R-tree Nodes

Fig. 1. Dynamic range query

• **Predictive**.— The system uses the user's motion parameters to predict his path (Q_i sequence) and uses the PDQ algorithm. Also used in "tour mode" in which the user's path is known a priori.
• **Non-Predictive**.— As the user's motion parameters change, the system uses the NPDQ algorithm until she settles down to a new direction/speed of motion; then PDQ takes over.

It must be noted that whether PDQ/NPDQ mode is used depends largely on the type of interaction in the system. A good direction of future research is to find automated ways to handle the PDQ $\leftrightarrow$ NPDQ hand-off. A third type of algorithm is also possible, that of *Semi-Predictive Dynamic Query* (SPDQ). In SPDQ, the trajectory of the user is allowed to deviate from the predicted trajectory by some $\delta(t)$, i.e., if the location of the observer at time t is $\bar{x}(t)$ and her predicted location is $\bar{x}_p(t)$, then SPDQ can be employed if $\| \bar{x}_p(t) - \bar{x}(t) \| \leq \delta(t)$. SPDQ can be easily implemented using the PDQ algorithms, but it will result in each snapshot query Q_i being "larger" than the corresponding simple PDQ one, allowing for the uncertainty of the observer's position.

4.1 Predictive Dynamic Queries

A predictive dynamic query (PDQ) is a dynamic query for which the database knows the trajectory of the observer and its associated range before executing it. In such cases, the query processor can partially precompute the answers for each snapshot of the dynamic query to optimize the cost. A predictive dynamic query is thus associated with a trajectory corresponding to that of the observer.

The trajectory of a PDQ query is captured by a sequence of key snapshot queries (as shown in Fig. 1) where a key snapshot query (K) is a spatial range window at a given time. That is, a trajectory of a PDQ query is identified with

key snapshot queries $K^1, \ldots, K^n$ where

$$K^j = \langle K^j.t, K^j.\overline{x}_1, \ldots, K^j.\overline{x}_d \rangle \wedge K^j.t < K^{j+1}.t \wedge j = 1, \ldots, n \qquad (2)$$

A PDQ query corresponds to the spatio-temporal region covered by joining all the key snapshot queries as illustrated in Fig. 1 (a). Two spatial dimensions are shown; you may visualize the observer moving from left to right in space as time (perpendicular to paper surface) progresses. Notice that, as shown in Fig. 1 (a), the application may ask numerous snapshot queries to the database in the context of a dynamic query in between two key snapshot queries. For example, in the visualization domain, the key snapshots correspond to the points of the fly-through defining the observer's trajectory. During the fly-through, numerous snapshot queries will be posed to the database, (one per each rendered frame). Our PDQ is different from the Type 3 (Moving Query) of [19] chiefly in that in [19] only a single segment of the trajectory is known (i.e., two key snapshots).

In Fig. 1 (b) a dynamic query is seen over the space indexed by an R-Tree. The query evolves with time from left to right. The vertical axis corresponds to the location of the observer; one spatial dimension is shown for simplicity. The query moves up or down along this vertical axis, as the observer moves in space. The query also becomes narrower, or broader, e.g., as the result of viewing the ground from different altitudes.

A simple approach to evaluating a PDQ query is to evaluate the set of answers for the entire duration of the query and return all these answers to the rendering system before they are needed. There are many problems with this approach:

- Since a PDQ query may cover a large spatial and temporal region with many objects in its path, the observer (rendering system in case of visualization example) must maintain a large buffer. In our approach, only currently visible objects need to be in the buffer.
- Even though the trajectory is considered predictive, an observer may deviate from the prescribed path of the trajectory. Such a deviation will result in wasted disk access since answers beyond the point of deviation may no longer be useful. If pre-computation of answers takes place and the observer deviates from her trajectory halfway through, then 50% of the expended effort was wasted. In our case, we can restart the PDQ algorithm if the trajectory changes; we don't retrieve objects before they are needed.
- Since objects being visualized are mobile and send location updates when they deviate from their motion information stored in the database, the location of those objects based on information at query evaluation time will not be valid. In our case, we retrieve objects when they are needed (late retrieval), giving them ample time to update their motion information, never reading the same object twice.

We propose an incremental approach to retrieve non-overlapping answers for evaluating dynamic queries. For each snapshot query asked, the database will only retrieve objects that have not been retrieved in previous snapshot queries. Along with each object returned, the database will inform the application about how long that object will stay in the view so that it will know how long the

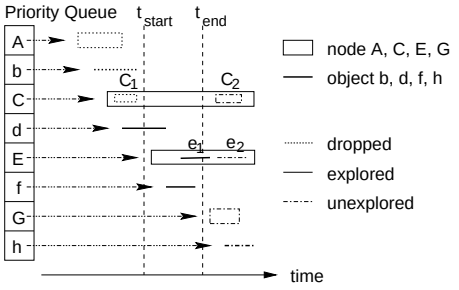


Fig. 2. Priority queue for dynamic query

object should be kept in the application's cache. For lack of space, we refer the reader to [14] for a more thorough study of caching issues. We only mention that it is easy (at the client) to maintain objects keyed on their "disappearance time", discarding them from the cache at that time.

The approach works as follows. Once the application has given the database all of the key snapshots, the database starts executing the PDQ by traversing the underlying multidimensional data structure. In describing the algorithm, we will assume that this structure is an R-tree though our technique works generally. We use a priority-based approach, similar to those of [17, 7] for kNN searches. Starting from the root node, the query processor computes the time interval that the root overlaps with the dynamic query and inserts it along with this interval in a priority queue, ordered by the start time of such time intervals. Note that if the time interval is empty, nothing is inserted in the queue. Next, for objects that will appear in view (i.e., be inside the query trajectory of Fig. 1) at time $t \in [t_{start}, t_{end}]$, repeated calls are made to $getNext(t_{start}, t_{end})$ (in Algorithm 4.1) until no object is returned or the priority queue is empty. Note that the function allows the application to query for objects that will appear in the view during time interval $[t_{start}, t_{end}]$. The time interval corresponds to the period between two key queries. By increasing t_{end} we would increase pre-fetching. We have already enumerated the reasons making this problematic without providing a benefit over our I/O optimal algorithm.

First, the query processor retrieves the top element in the priority queue. or $\varnothing$ if it is empty. Otherwise, it keeps reading items from the queue and examining their children by computing the time interval that each child will overlap with the query. Then, each child will be inserted along with its time interval in the queue. This step goes on until an object is found at the top of the queue with an associated time interval overlapping with t. If such an object is found, it is returned. If the time interval of the item at the head of the queue is greater than t (future with respect to t) or the queue is empty, $\varnothing$ is returned. Fig. 2 illustrates how the algorithm works. Node A and object b are at the head of the queue and will be dropped off the queue. Node C will be explored and C_1 will be dropped while C_2 will be inserted to the queue. Object d is the first object that will be returned. Subsequent calls to $getNext()$ (with the same arguments) will explore

Pseudo-code 4.1 getNext() function

```
1  item function getNext(t_start, t_end)
2  {
3     item = PriorityQueue.peek();
4     if (item == ∅) return ∅;
5     while (t_end ≥ item.time_start) {
6        item = PriorityQueue.pop();
7        if (t_start ≤ item.time_end) {
8           if (item.isObject()) return item;
9           RTree.loadNode(item); // disk access
10          for each (i ∈ item.children) {
11             i.time = Query.computeOverlappingTime(i);
12             if (t_start ≤ i.time_end) PriorityQueue.push(i);
13          }
14       }
15       item = PriorityQueue.peek();
16       if (item == ∅) return ∅;
17    }
18    return ∅;
19 }
```

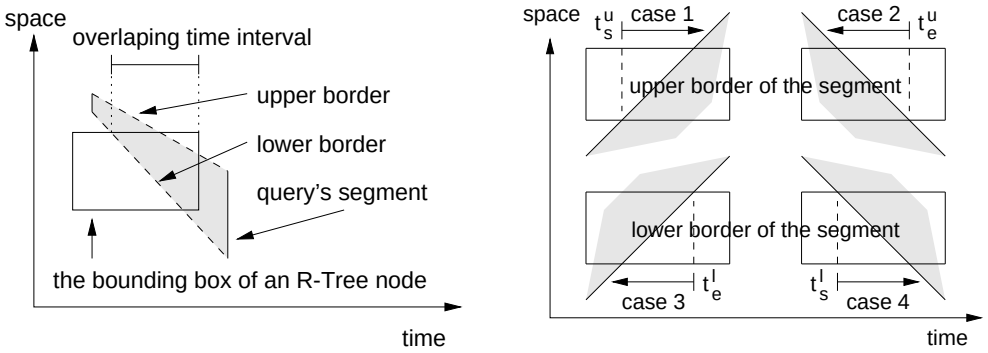
node E, insert e_1 and e_2 to the queue, and return f and e_1 eventually. Node G, node C_2, object e_2, and object h will remain in the queue until the application calls the function with different argument values.

This algorithm guarantees that the first object appearing during $[t_{start}, t_{end}]$ will be returned. In Fig. 2, objects d, f, and e_1 will be returned in that order.

We compute the time interval that a bounding box R will overlap with the dynamic query by identifying the subsequence of key snapshots (K) that temporally overlap with the bounding box. For each pair of consecutive key snapshots in the subsequence, we compute the time interval T^j that the trajectory between K^j and K^{j+1} overlaps with R. We call the trajectory between K^j and K^{j+1} a segment S^j. Each segment has a trapezoid shape as shown in Fig. 3 where a single spatial dimension (vertical axis) and the temporal dimension (horizontal) is shown. Let R be $\langle \overline{t}, \overline{x}_1, \ldots, \overline{x}_d \rangle$. We compute T^j as follows:

$$T^j = \bigcap_{i=1}^{d} T_i^{j,u} \cap T_i^{j,l} \cap [K^j.t, K^{j+1}.t] \cap R.\overline{t} \tag{3}$$

where $T_i^{j,u} = [t_{i,s}^{j,u}, t_{i,e}^{j,u}]$ is the time interval that the upper border of S^j overlaps with R along the i^{th} dimension and $T_i^{j,l} = [t_{i,s}^{j,l}, t_{i,e}^{j,l}]$ is the time interval that the lower border of S^j overlaps with R along the i^{th} dimension (see Fig. 3 (a)).



(a) Overlapping Time Interval

(b) Intersection of a Bounding Box and a Segment

Fig. 3. Overlapping time interval

We compute $T_i^{j,u}$ and $T_i^{j,l}$ by checking four cases as shown in Fig. 3 (b). For example, in Case 1, where the upper border of the segment moves upward in time, t_s^u is the intersection of the upper border of the segment and the lower border of the bounding box and t_e^u is the ending time of the bounding box.

For each S^j with $j = 1, \ldots, n-1$, we compute T^j. Next, we compute the overlapping time interval of the dynamic query and a bounding box $T_{Q,R}$ as $\uplus_{j=1}^{n-1} T^j$. We insert $T_{Q,R}$ along with the node corresponding to R in the queue.

For the leaf node where motions are stored, similar to the bounding boxes in internal nodes, we can compute $T_i^{j,u}$ and $T_i^{j,l}$ by checking the four cases. The geometric intuition for computing the intersection of both motion segments and bounding boxes with the query trajectory is fairly simple. To conserve space, we give the precise formulae in [9].

The benefit of this over the naive approach (in which each snapshot query is independently evaluated) is that we access each R-tree node at most once irrespective of the frame rate (snapshot rate) that the application attempts to render. In the naive approach, each frame corresponds to a snapshot query and thus the disk access cost is proportional to the frame rate.

Update Management: The described mechanism works in a situation where there are no motion updates submitted to the R-Tree while the query is running. Such a situation is common in historical databases. If updates occur concurrently with dynamic queries, a special mechanism is needed to handle them. Each such update is in fact an insertion operation, resulting in the insertion of a new line-segment object into the R-tree. All ongoing dynamic queries must be aware of newly inserted objects. For each inserted object, the query processor checks whether it will appear in any dynamic queries, and inserts it in the queues of such queries. Similarly, if a node overflows, a new node is created to share the overflow node's load. We check whether this new node overlaps with

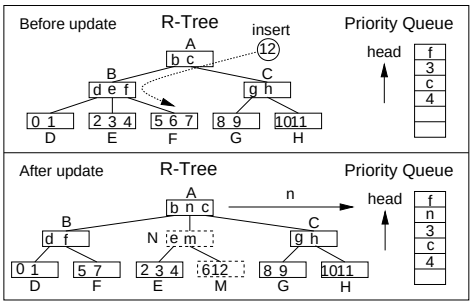


Fig. 4. Update management for PDQ

any current dynamic queries. If so, then we insert it in the queue of that query. This ensures that dynamic queries are aware of new branches in the R-tree.

Note that when a leaf node overflows, a sequence of overflows and splits all the way to the root node may occur. The original R-tree insertion algorithm [5] did not guarantee that all newly created nodes will be in the same path. However, it is possible to force them to be on the same path as the data causing the overflow. Doing so incurs no extra cost nor conflict with the original splitting policy. The benefit of this is that only the lowest-common ancestor of the newly created nodes needs to be inserted into the priority queue; this covers all the new nodes (including the inserted data). Figure 4 demonstrates such a scenario. When the object with ID 12 is inserted in node F, node F overflows and splits to node M. Node M is then inserted into node B, which causes node B to overflow and split to node N. Node N is then inserted to node A. Since node A is not split further, N is the highest node of the sequence (12, M, N), n is sent to all dynamic queries for consideration to be inserted into their priority queues.

Notice that, before an update occurs, for each node in the priority queue, it is impossible for its ancestors to be in the same queue. This is because a node cannot be in the queue unless its parent has been explored. A node that has been explored won't be put back in the queue. In other words, the priority queue will never explore any node more than once. However, unfortunately, this is not true when the update mechanism is involved. For example, in Fig. 4, node N is an ancestor of object ID 3 and 4 which are also in the queue. When node N is explored, node E and M will be inserted into the queue and eventually object ID 3 and 4 will be inserted into the queue again. Although, we cannot prevent duplicate insertions into the queue, we can eliminate the duplication when popping nodes from the queue. Although duplicate items are inserted in the queue at different times, they are popped out of the queue consecutively since they have the same priority. Thus, each time that we pop an item from the queue, we check if it is the same as the previous one[2]. If yes, we just ignore that item since it has just been taken care of.

[2] More generally, we check a few objects with the same priority, since two objects may appear concurrently

If the lowest common ancestor of newly created nodes is close to the root node, it is better to empty the priority queues of the current dynamic queries and rebuild the queue from the root node since there is a high possibility of duplicate insertion. Similar to the growth rate of the height of the R-tree, the rate that such a node will become close to the root node is exponentially decreasing.

4.2 Non-predictive Dynamic Queries

Predictive dynamic queries described in the previous section take advantage of the known trajectory of the query and precompute the search path. In some cases, it is not possible to know the trajectory beforehand. The techniques to evaluate non-predictive dynamic queries are designed for this purpose.

A non-predictive dynamic query is a dynamic query for which the database does not know the trajectory nor the range of the query beforehand. As a result, the query processor cannot precompute the answers or the search path prior to receiving each snapshot query. However, the query processor remembers the previously executed query, and thus can still avoid repeating *some* of the work previously done. Similar to PDQ, NPDQ returns only additional answers that have not been returned in the previous query. Like PDQ, since the query rate is high, there may be significant overlap between consecutive queries. The naive approach (independent evaluation of snapshot queries) will visit nodes of the data structure multiple times retrieving the same objects repeatedly. The approach described below overcomes this problem.

Formally, we define a non-predictive query (NPDQ) as a pair of snapshot queries P and Q where $P.t < Q.t$ and the objective is to retrieve all objects that satisfy Q and have not been retrieved by P. The first snapshot query of NPDQ is evaluated as a normal snapshot query. For subsequent queries, the query processor will check each bounding box (starting from that of the root node) if it is **discardable**. A bounding box R is discardable iff the overlapping part of R and Q is covered by P. Otherwise, the node corresponding to the box R needs to be explored. The following lemma (proven in [9]) formalizes the condition when a given node R of a data structure is discardable.

Lemma 1 (Discardable). *Let Q be the current query box and P be the previous one. Let R be a bounding box corresponding to an R-tree node. R is* **discardable** *iff $(Q \cap R) \subset P$.*

The search algorithm for NPDQ works as follows. The algorithm starts by visiting the root node of the R-tree. For each child R of the currently visited node, R is visited if $discardable(P, Q, R)$ is false where $discardable(P, Q, R)$ has the truth value of $(Q \cap R) \subset P$. In dynamic queries, notice that P and Q never overlap along the temporal axis since P extends from time t_i to t_j and Q extends from t_j to t_k where $t_i < t_j < t_k$. This implies that if $(Q \cap R)$ is not empty, $(Q \cap R) \subset P$ is always false. That is, *discardability* is useless since the previous query does not reduce the number of nodes that need to be visited in the current one. To overcome this problem we can: (i) use an open-ended temporal range query (Fig. 5(a)), or (ii) use double-temporal axes (Fig. 5(b)).

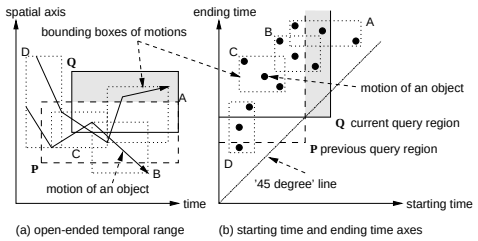


Fig. 5. Discardability

Using an open-ended temporal range query can overcome the problem since the previous query retrieves all objects which satisfy the spatial range of the query either now or in the future. This is suitable for querying future or recent past motions only. See Fig. 5(a): B and C do not need to be visited by the current query Q since they have already been visited by the previous query P. D corresponds to a node that Q does not need to visit since it does not overlap with Q. However, Q needs to visit A because A may contain an object which does not overlap with P, and was thus not retrieved in the previous query.

Another approach to overcome the same problem is to separate the starting time and the ending time of motions into independent axes. See Fig. 5(b): all objects reside on one side of the 45 degree line since all motions start before they end. A and B are BBs that Q needs to visit while C and D are BBs that P has visited and Q does not need to visit. This approach is suitable for both historical and future queries. We choose this approach in our implementation of NPDQ. We give a formal definition of the $discardable(P, Q, R)$ condition in [9].

Update Management: We use a timestamp mechanism to handle updates (i.e., insertions of new motions). For each insertion, all nodes along the insertion path will update their timestamp to the current time. This lets the search algorithm know if the discardability condition is to be employed. When a BB is checked during the search algorithm, initially its timestamp is read; if it is newer than that of the previous query P then R has been modified since the last visit and P cannot be used to eliminate R for Q. Thus, the normal condition $(R \not\between Q)$ is employed. If $P.t$ is not older than the timestamp of R, then R has not changed since the last visit; P can be used to check if R is discardable for query Q.

5 Empirical Evaluation

We studied the performance of our DQ algorithms (for both PDQs and NPDQs), comparing them to the naive approach of multiple instantaneous queries. Our performance measures are I/O cost measured in number of disk accesses/query and CPU utilization in terms of number of distance computations.

Data and Index Buildup: We generate data as follows: 5000 objects are created, moving randomly in a 2-d space of size 100-by-100 length units, updating their motion approximately (random variable, normally distributed) every 1 time

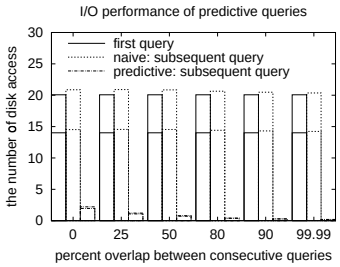


Fig. 6. I/O performance of PDQ

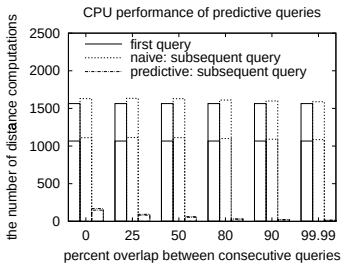


Fig. 7. CPU performance of PDQ

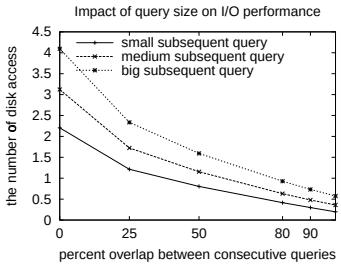


Fig. 8. Impact of Query Size on I/O Performance of Subsequent Queries (PDQ)

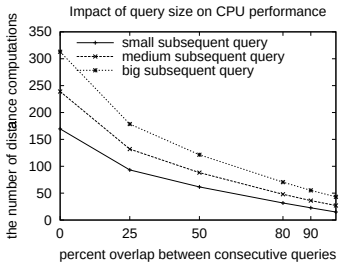


Fig. 9. Impact of Query Size on CPU Performance of Subsequent Queries (PDQ)

unit over a time period of 100 time units. This results in 502,504 linear motion segments being generated. Each object moves in various directions with a speed of approximately 1 length unit/1 time unit. Experiments at higher speeds were also run with similar results. Plotted figures correspond to the first case. An R-tree, NSI index is used to store this motion data as discussed in Section 3.

Page size is 4KB with a 0.5 fill factor for both internal and leaf nodes. Fanout is 145 and 127 for internal– and leaf-level nodes respectively; tree height is 3.

Queries: To compare performance of our approach with the naive one, 1000 query trajectories of each speed of the query trajectory are generated and results are averaged. Query performance is measured at various speeds of the query trajectory. For each DQ, a snapshot query is generated every 0.1 time unit. For a high speed query, the overlap between consecutive snapshot queries is low; this increases as speed decreases. We measure the query performance at overlap levels of 0, 25, 50, 80, 90, and 99.99%. We also measure the impact of the spatial range of the query on the performance of the dynamic query. We use the spatial range of 8x8 (small range), 14x14 (medium range), and 20x20 (big range). Unless otherwise stated, the experiments reported are based on the small range query.

Experiments for PDQ: Figure 6 plots the total number of disk accesses (including the fraction of them at the leaf level) for different percent overlap between consecutive snapshot queries. For each histogram bar, the lower part of the bar shows the number of disk accesses at the leaf level while the upper part of the bar shows the number of disk accesses at higher levels of the R-tree. The figure

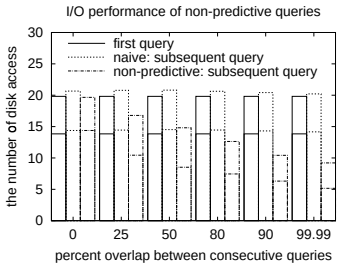


Fig. 10. I/O performance of non-predictive dynamic query

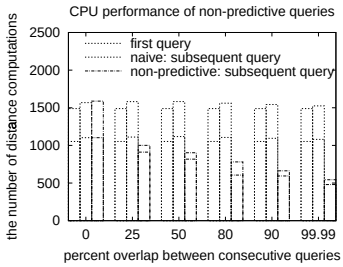


Fig. 11. CPU performance of non-predictive dynamic query

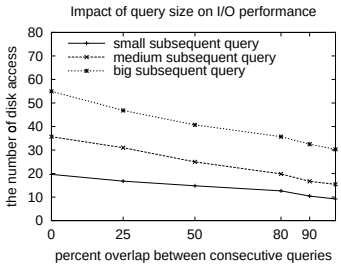


Fig. 12. Impact of query size on I/O performance of subsequent queries (NPDQ)

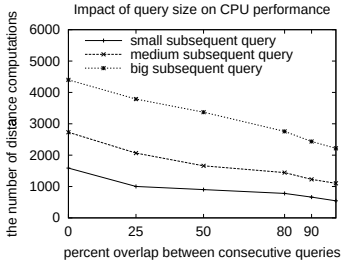


Fig. 13. Impact of query range on CPU performance of subsequent queries (NPDQ)

shows the I/O performance of the first query and subsequent queries. The results of subsequent queries are averaged over 50 consecutive queries of each dynamic query. The figure shows that, in the naive approach, the query performance of subsequent queries is the same as that of the first snapshot query and independent of the percent overlap between the consecutive snapshot queries. This is because the naive approach re-executes each snapshot query from scratch. The proposed approach to predictive query significantly improves the query performance for subsequent queries. The more the percent overlap is, the better I/O performance is. Even in the case of no overlap between subsequent queries, the predictive approach still improves the query performance significantly. This is because the predictive approach benefits from spatio-temporal proximity between consecutive queries. Similar to Fig. 6, Fig. 7 shows the CPU performance of predictive queries in term of the number of distance computation. The number of distance computations is proportional to the number of disk accesses since, for each node loaded, all its children are examined. So, Fig. 7 is similar to Fig. 6.

Figures 8, 9 show the impact of spatial range of the dynamic query on the I/O and CPU performance respectively. The figures show the performance of the subsequent queries of the dynamic query. The results are intuitive in that a big query range requires a higher number of disk accesses and a higher number of distance computations as compared as opposed to a smaller one.

Experiments for NPDQ: Similar to the results for PDQ, Fig. 10 plots the total number of disk accesses (including the fraction of them at the leaf level) for varying degree of overlap between consecutive snapshot queries. Here as well, NPDQ improves performance significantly for subsequent queries. As overlap increases, I/O performance improves. If there is no overlap between two consecutive queries, the NPDQ algorithm does not cause improvement; neither does it cause harm. Fig. 11 shows the CPU performance of NPDQ, similar to the result for I/O shown in Fig. 10. Similarly to PDQ, Figures 12, Fig. 13 indicate that the size of the spatial range of the query impacts its I/O and CPU performance. That is, increase in size results in more disk I/O and a higher number of distance computations. Comparison of PDQ versus NPDQ performance favors the former; this is expected due to the extra knowledge being used.

6 Conclusion

In this paper, we presented efficient algorithms to evaluate *Dynamic Queries* in database systems. We have dealt with two types of such queries: *predictive* and *non-predictive* ones (PDQ/NPDQ). In PDQ we utilize knowledge of the query's trajectory and precompute answers as needed. NPDQ is the general case in which the query trajectory is not known beforehand and thus answers can't be precomputed. However, since some of the query results from previous snapshot queries can be re-used, the algorithm for NPDQ avoids performing multiple disk accesses. An intermediate case, Semi-Predictive Dynamic Query (SPDQ) in which the query trajectory is known approximately within an error bound, was also explored. Our experimental study indicates that both predictive and non-predictive queries resulted in significant improvement of retrieval performance.

There are many directions of future research: (i) generalizing the concept of dynamic queries to nearest neighbor searches as well, similar to moving-query point of [24], (ii) generalizing dynamic queries to include more complex queries involving simple or distance-joins and aggregation, (iii) adapting dynamic queries to a specialized index for mobile objects such as TPR-tree, [19] (iv) investigating the spectrum of possibilities between complete unpredictability and complete predictability of query motion and automating this in the query processor.

References

1. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.
2. N. Beckmann, H.-P. Kriegel, R. Schneider, and B.Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD*, 1990.
3. K. Chakrabarti, K. Porkaew, and S. Mehrotra. Efficient query refinement in multimedia databases. In *IEEE ICDE 2000*.
4. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databa ses. In *ACM SIGMOD*, 2000.
5. A. Guttman. R-tree: a dynamic index structure for spatial searching. In *ACM SIGMOD*, 1984.

6. G. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *ACM SIGMOD*, 1998.
7. G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Int'l Symposium on Large Spatial Databases*, 1995.
8. G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Symposium on Principles of Database Systems*, 1999.
9. I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. (full version) Technical Report TR-DB-01-07, UC Irvine, 2001.
10. D. Lomet and B. Salzberg. The hB-Tree: A multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, 1990.
11. D. Lomet and B. Salzberg. The performance of a multiversion access method. In *ACM SIGMOD*, 1990.
12. D. Pfoser and C. Jensen. Capturing the uncertainty of moving-object representations. In *SSD*, 1999.
13. D. Pfoser, C. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *VLDB*, 2000.
14. K. Porkaew. Database support for similarity retrieval and querying mobile objects. Technical report, PhD thesis, University of Illinois at Urbana-Champaign, 2000.
15. K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *SSTD 2001, Redondo Beach, CA, USA*.
16. J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *ACM SIGMOD*, 1981.
17. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *ACM SIGMOD*, 1995.
18. S. Saltenis and C. Jensen. Indexing of moving objects for location-based services. In *ICDE (to appear)*, 2002.
19. S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD*, 2000.
20. B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.
21. H. Samet. The quadtree and related hierarchial data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.
22. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+ tree: A dynamic index for multi-dimensional objects. In *VLDB*, 1987.
23. A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *IEEE ICDE 1997*.
24. Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, 2001.
25. J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree based dynamic attribute indexing method. *Computer Journal*, 41(3):185–200, 1998.
26. D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *ACM SIGMOD*, 1992.
27. Y. Theodoridis, T. Sellis, A. N. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *SSDBM*, 1998.
28. O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *IEEE ICDE*, 1998.