Querying Mobile Objects in Spatio-Temporal Databases

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

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

Abstract. In dynamic spatio-temporal environments where objects may continuously move in space, maintaining consistent information about the location of objects and processing motion-specific queries is a challenging problem. In this paper, we focus on indexing and query processing techniques for mobile objects. Specifically, we develop a classification of different types of selection queries that arise in mobile environments and explore efficient algorithms to evaluate them. Query processing algorithms are developed for both native space and parametric space indexing techniques. A performance study compares the two indexing strategies for different types of queries.

1 Introduction

Increasingly, emerging application domains require database management systems to represent mobile objects and support querying on the motion properties of objects [16,5,12]. For example, a traffic monitoring application may require storage of information about mobile vehicles (e.g., buses, taxicabs, ambulances, automobiles). Similarly, tanks, airplanes, and military formations in a war simulation application need to be represented. A weather simulation would need representation and querying over trajectories of various weather patterns, e.g., storm fronts. Motion properties of objects are *dynamic* – i.e., the location of mobile objects changes continuously as time passes, without an explicit update to the database[1]. This is in contrast to *static* objects traditionally supported in databases whose value changes only with an explicit update.

Motion information of objects can be represented in the database using *motion parameters* and *location functions* which compute the position of the object in space at any time. Different such parameters and functions may be associated with different objects, based on the type of their motion. As an example, consider an object that translates linearly with constant velocity in 2-dimensional space. The motion parameters in this case correspond to the object's starting location (x_s, y_s) at some initial time t_s and its velocity (v_x, v_y) along the spatial dimensions. Using these parameters the location of the object at any future time $t > t_s$ can be determined.

[1] Other object properties besides location can also be dynamic, e.g., fuel level of a moving vehicle

C.S. Jensen et al. (Eds.): SSTD 2001, LNCS 2121, pp. 59–78, 2001.

Types of Queries: Similar to queries in traditional spatio-temporal databases over static objects, queries over mobile objects involve projection, selection, join, and aggregation operations. In this paper, we focus on *selection queries* since they are the building block for more complex SQL queries. Selection queries may be classified as either *range* or *nearest neighbor* queries. In the first case, objects that fall within a given spatio-temporal range are retrieved. In the second one, we are interested in objects that are relatively "closer" to the query point.

The notion of *proximity* differs between time and the spatial dimensions. In the temporal sense, proximity means co-occurence within a certain time period. Spatially, it refers to geographical/geometrical closeness. Thus, it is highly desirable to separate the selection predicate into spatial and temporal components, each of which can be specified in a different manner, either as a k-Nearest Neighbor or Range predicate.

In Fig. 1 the types of queries that are meaningful are marked with ($\surd$). We will now give examples of queries handled in our system.

	Temporal **kNN**	Temporal **Range**
Spatial **kNN**	$\times$	$\surd$
Spatial **Range**	$\surd$	$\surd$

Fig. 1. Spatio-Temporal Selection Query Classification

• A **Spatio-temporal Range Query** is specified by both a spatial and a temporal range. An example is "retrieve all vehicles that were *within a mile of the location* (spatial range) of an accident *between 4-5pm* (temporal range)".

• A **Temporal kNN Query** is specified with a spatial range and a nearest neighbor predicate on the temporal value. An example is "retrieve the first ten vehicles that were *within 100 yards* (spatial range) from the scene of an accident ordered based on *the difference between the time the accident occured and the time the vehicle crossed the spatial region* (temporal kNN predicate)". Note that in a temporal kNN query, we are sometimes interested only in the most recent past or in upcoming future events but not in both. In such cases, the temporal kNN query will expand the search only in one direction of the temporal dimension.

• A **Spatial kNN Query** is specified with a temporal range and a kNN predicate on the spatial dimensions. For example, "retrieve the five ambulances that were *nearest to the location* of the accident *between 4-5pm*".

• A **Spatio-Temporal kNN Query**, marked with ($\times$) in Fig. 1, would specify a kNN predicate in both the spatial and temporal dimensions. Such a query type is not possible since it would imply that a total order existed between the temporal and the spatial dimensions.

Contributions: We explore two approaches – *native space indexing* (NSI) and *parametric space indexing* (PSI) and we have developed algorithms for evaluating all previously described query types, using both of these approaches. In the NSI approach, motion is indexed in the original space in which it occurs, while in the PSI approach a parametric space defined by the motion parameters of the objects is used. Work on indexing mobile objects has recently been studied in [16,5,12]; we will discuss the state of the art in the following section. This paper makes the following contribution to the existing literature. First, while most existing approaches have explored techniques to support spatio-temporal range queries over mobile objects, we also develop techniques for temporal and spatial kNN queries as described above, by appropriately adapting techniques proposed for kNN in [11,4]. Furthermore, the emphasis of existing techniques [5,12] has been on the PSI strategy. In contrast, we explore both NSI and PSI strategies and compare them for different types of queries. Our results actuallly illustrate that NSI almost always performs better compared to PSI strategies for all query types. We provide an intuitive justification for this behavior.

Roadmap of the Paper: The rest of the paper is organized as follows. Sect. 2 discusses related work on indexing of motion. Sect. 3 discusses two motion indexing techniques (native space- and parametric space- indexing) and corresponding query processing techniques for various query types. In Sect. 4 we evaluate these techniques. Sect. 5 concludes the paper.

2 Related Work

Techniques for indexing multidimensional data (including spatial, temporal, and spatio-temporal data) have been extensively studied in the literature. Example data structures developed include R-tree and its family [3,2,15], Quadtree [14], and hB-tree [6]. Specialized techniques to index temporal objects have also been proposed, including multi-version index structures, e.g., Time-Split B-tree (TSB-tree) [7], multi-version B-tree [1], and others, summarized in [13]. While the focus of these data structures has been on indexing temporally changing attribute values, they can be adapted to index spatio-temporal objects by replacing the key attribute (that is indexed using B-tree) by the spatial location, which will be indexed by a spatial data structure.

Existing proposals for spatio-temporal index structures have been summarized in [17]. The primary difference between the multi-version index structures and the spatio-temporal index structures is that in a multi-version index the temporal dimension is differentiated from other dimensions. The splitting policy focuses on temporal split with the objective to keep the "live" data which is currently active clustered together. In contrast, spatio-temporal index structures do not differentiate between the spatial and temporal dimensions.

The focus of the above described spatial, temporal, and spatio-temporal data structures has been on representing static objects – ones whose value changes

only with an explicit update to the data structure. Some recent papers have explored indexing issues for dynamic properties – that is, properties whose value changes continuously as time passes without an explicit update. Specifically, research has focussed on indexing and query processing over mobile data [16,5,12]. Most of this work has concentrated on techniques to support spatio-temporal range queries based on current motion properties of objects projected onto a time in the future. For example, in [16], Quadtree is used to partition the embedded space in which object motion occurs. Mobile objects are represented by clipping their motion trajectories into the collection of leaf cells that they intersect. Standard techniques are then used to support spatio-temporal range queries.

In [5], motion is represented in a parametric space using the velocity of the object and its projected location along each dimension at a global time reference. Given a parametric representation of motion, a standard multidimensional index structure (specifically, kDB-tree [10]) is used with dimensions corresponding to velocity and location. The disadvantage of indexing positions projected at a global time reference is a loss of *locality*. By this, we mean that the relative placement of objects' projections at a global time reference does not carry much information as to their location at the present or future times. The authors also suggest an improvement to their approach by transforming and projecting 2-d parametric space (i.e., velocity and location dimensions) onto 1-d space and use a B-tree to index 1-d data. This technique is, however, applicable only to motion in 1-d space unless multiple B-trees are used (one per spatial dimension) for multidimensional motion. Furthermore, the projection of 2-d parametric space corresponding to the velocity and location to a single dimension also sacrifices the accuracy of the answer.

In [12], similar to [5], motion is represented in a parametric space defined by its velocity and projected location along each spatial dimension at a global time reference. The parametric space is then indexed by a new index structure referred to as the TPR-tree (Time-Parameterized R-tree). TPR-tree is similar to R*-tree [2] except that TPR-tree does not represent a node's boundary with a bounding box. Rather, a node boundary of TPR-tree corresponds to an open-ended trapezoidal shape in the native space where the parallel sides are perpendicular to the temporal axis. The splitting algorithm splits an overflow node into two nodes whose corresponding trapezoids are minimized in size (or volume). The search algorithm for a range query also performs computation on the native space by checking the overlap between the range of the query and the trapezoid representation of the node. The TPR-tree uses an optimization that attempts to compact the spatial coverage of the leaf nodes (resulting in improvement of query performance). However, the optimization can be used if the data structure is used to answer only future queries.

3 Indexing and Query Processing of Motion

3.1 Model and Assumptions

We represent motion of objects using motion parameters and location functions. A *location function* is a generic method by which the position of an object at any valid time can be deduced and it corresponds to a type of motion, e.g., linear translation. *Motion parameters* specify the instantiation of the location function, e.g., a starting location and speed of motion in the linear translation example introduced in Sect. 1. Associated with each object's motion is a valid time interval $\bar{t} = [t_{\mathrm{l}}, t_{\mathrm{h}}]$ when the motion is valid. The location functions can be used to compute the position of the object at any time $t' \in \bar{t}$ in a d-dimensional space. In this paper, we are primarily concerned with object trajectories that correspond to linear motion with constant speed. The following location function determines the location of an object O at time t':

$$O.X_i(t') = O.x_i + O.v_i(t' - O.t_{\mathrm{l}}) \text{ where } i = 1, 2, ..., d \text{ and } t' \in [O.t_{\mathrm{l}}, O.t_{\mathrm{h}}] \quad (1)$$

where $O.x_i$ is the location and $O.v_i$ is the velocity of the object along the i^{th} dimension respectively at the starting time $O.t_{\mathrm{l}}$, for all $i = 1, \ldots, d$.

As time progresses, the motion of an object may deviate from its representation in the database (as the vector of its velocity changes). When the deviation exceeds a threshold, an object (or sensors tracking the object) updates its motion parameters and/or the location function stored in the database to reflect its most current motion information. Each update contains the location $O.x_{1...\mathrm{d}}$, the velocity $O.v_{1...\mathrm{d}}$, the time of the update $O.t_{\mathrm{l}}$ and the time of the next update (expiration time of this update), $O.t_{\mathrm{h}}$.

The expiration time can be viewed as a promise to issue an update no later than $O.t_{\mathrm{h}}$. If an update is issued at time $t_{\mathrm{new}} < O.t_{\mathrm{h}}$, then the new motion information takes precedence over the previous one for the interval $[t_{\mathrm{new}}, O.t_{\mathrm{h}}]$ in which they are both valid. In practical terms, this means either that the new update replaces the old one (no historical information is maintained), or that the two co-exist and thus for a given time, the object might be represented (and thus retrieved) by more than one database object. Thus, it would be necessary to process objects after retrieval to discard duplicates by retaining the most recent update.

Note that there is a tradeoff between the cost of update (which depends upon the update frequency) and the precision of information captured by the database. The update management issues including the above tradeoff have been discussed previously in [18]. If the imprecision of an object's representation is taken into account, its location at a given instance of time, instead of corresponding to a point in space (as suggested by the location function above), will rather be a bounded region that represents the *potential* location of an object. For simplicity of exposition, we will assume that the parameters used to capture the motion of an object in the database precisely determine its location until the next update of the object's motion. The mechanisms for indexing and query processing

proposed can be extended straightforwardly to the case when the imprecision in the location of an object is taken into account (that is, the location of an object at each instance of time, instead of corresponding to a point, corresponds to a bounded region).

To evaluate the spatio-temporal selection queries, we do not wish to scan through the whole database to check the query predicate for each object. Instead, for efficiency, we index these objects in a multidimensional data structure and utilize the data structure to identify the objects that satisfy the query predicate. There are many ways to index the moving objects. In general, we can classify indexing mechanisms into two types: **Native Space Indexing** (NSI) and **Parametric Space Indexing** (PSI) which are discussed next. In the discussion, the following notation will be used:

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 are intersection ($\cap$) and overlap ($\between$). Let $\overline{J} = [J_\mathrm{l}, J_\mathrm{h}]$ and $\overline{K} = [K_\mathrm{l}, K_\mathrm{h}]$. $\overline{J} \cap \overline{K} = [\max(J_\mathrm{l}, K_\mathrm{l}), \min(J_\mathrm{h}, K_\mathrm{h})]$. $\overline{J} \between \overline{K}$ returns the boolean value of $\overline{J} \cap \overline{K} \neq \varnothing$.*

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 $I_1 \times I_2 \times \ldots \times I_n \subset \Re^d$. A box $\square B$ is an empty box ($\square B = \varnothing \Leftrightarrow \exists i : \overline{I}_i = \varnothing$). An n-dimensional point $p = \langle v_{1\ldots n} \rangle$ is equivalent to box $\big\langle [v_1, v_1], [v_2, v_2], \ldots, [v_n, v_n] \big\rangle$. Operations on boxes are the same as those on intervals. Their meanings are intuitive and similar to those of intervals.*

3.2 Native Space Indexing Technique (NSI)

Consider an object O whose motion is captured by a location function $O.X_{1\ldots\mathrm{d}}(t)$. Let the valid time for the motion of the object be $O.\bar{t}$. In NSI, the motion of O is represented with a bounding box with extents $[\min_{t \in O.\bar{t}} O.X_i(t), \max_{t \in O.\bar{t}} O.X_i(t)]$ along each spatial dimension i, and extent $O.\bar{t}$ along the temporal dimension. A multidimensional data structure (e.g., R-tree) is used to index the bounding box representation of motion. Note that the index will contain multiple bounding boxes to represent the motion trajectory of an object, one per motion update. We next discuss how different types of queries are evaluated in the NSI strategy. While the algorithms discussed below are applicable for any multidimensional data structure, in the following discussion, we will assume that an R-tree is used to index motion.

Spatio-Temporal Range Queries: Evaluating range queries in NSI is straightforward. The standard R-tree range search algorithm is used to traverse the data structure for objects whose motion trajectory overlaps with the query. The search begins from the root and all children whose bounding boxes overlap with the query region are traversed. The bounding boxes stored in the leaf node

represent objects' boundaries. If these bounding boxes overlap with the query, the corresponding object is retrieved.

Note that it is quite possible that even though an object's motion trajectory does not overlap with the query, the bounding box corresponding to motion does overlap, resulting in false admissions. The cost of such false admissions is quite significant since each such admission may result in one extra disk access to retrieve an object that will immediately be discarded.

The problem of false admissions in the NSI strategy can be eliminated by representing motion as a line segment instead of a bounding box at the leaf level of the R-tree. The representation of a line segment is similar to that of a bounding box. That is, a bounding box is represented by its lower left corner point and its upper right corner point while a line segment is represented by its starting point and its ending point. Detecting an overlap between a query and a line segment object is more complicated than detecting an overlap between a query and a bounding box object.

To detect if there is an overlap between query's bounding box (Q) and the line segment corresponding to the motion of an object (O) we compute the time interval that the motion trajectory overlaps with the query bounding box along each spatial dimension i separately and then intersect the computed intervals. If the intersection is empty, then the motion does not overlap with the query's bounding box and thus the query can ignore that motion. Otherwise, the object is returned as an answer.

Formally, let $T_{\mathrm{Q,O}}$ be the time interval that the motion overlaps with the query's bounding box and $T_{\mathrm{Q,O},i}$ be the time interval that the motion overlaps the query's bounding box along dimension i. We compute $T_{\mathrm{Q,O},i}$ as follows. Considering only one spatial dimension with the temporal dimension, as shown in Fig. 2, we extend the line segment corresponding to the motion at both ends infinitely. Next, we compute the time $t_{i,\mathrm{s}}$ and $t_{i,\mathrm{e}}$ which this infinite line cuts the lower border (i.e., the line equation $x_i = Q.x_{i\mathrm{l}}$) and upper border (i.e., the line equation $x_i = Q.x_{i\mathrm{h}}$) of the query's bounding box along spatial dimension i. Then, we compute $T_{\mathrm{Q,O},i}$ as the intersection of the query's time interval ($[Q.t_{\mathrm{s}}, Q.t_{\mathrm{e}}]$), the motion's time interval ($[O.t_{\mathrm{s}}, O.t_{\mathrm{e}}]$) and $[t_{i,\mathrm{s}}, t_{i,\mathrm{e}}]$. That is,

$$T_{\mathrm{Q,O}} = \cap_{i=1}^{d} T_{\mathrm{Q,O},i} \quad \text{where} \quad T_{\mathrm{Q,O},i} = [t_{i,\mathrm{s}}, t_{i,\mathrm{e}}] \cap [O.t_{\mathrm{s}}, O.t_{\mathrm{e}}] \cap [Q.t_{\mathrm{l}}, Q.t_{\mathrm{h}}] \quad (2)$$

where d is the number of spatial dimensions. In case the motion along spatial dimension i is parallel to the temporal axis (when there is no movement along that spatial axis), the motion intersects with the query if the motion is in between the lower and the upper borders of the query's bounding box. Notice that the motion can never be perpendicular to the temporal axis since an object cannot be at multiple locations at the same time. The complexity of this detection is $O(d)$, where d is the number of the spatial dimensions, which is the same as the complexity of detecting the overlap of two bounding boxes.

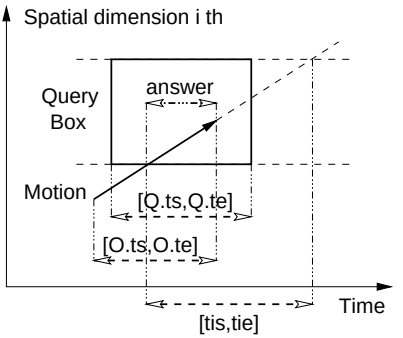


Fig. 2. Temporal Intersection of Object Motion and Query Bounding Box

We compute $t_{i,\mathrm{s}}$ and $t_{i,\mathrm{e}}$ by finding the intersection of the upper bound $Q.x_{i\mathrm{h}}$ (and lower bound $Q.x_{i\mathrm{l}}$) of the bounding box query and the line that connects between $(O.t_{\mathrm{s}}, O.x_{i\mathrm{s}})$ and $(O.t_{\mathrm{e}}, O.x_{i\mathrm{e}})$ as follows:

$$t_{i,\mathrm{s}} = \begin{cases} \dfrac{Q.x_{i\mathrm{l}} - O.x_{i\mathrm{s}}}{O.x_{i\mathrm{e}} - O.x_{i\mathrm{s}}}(O.t_{\mathrm{e}} - O.t_{\mathrm{s}}) + O.t_{\mathrm{s}} & \text{if } O.x_{i\mathrm{s}} < O.x_{i\mathrm{e}} \\ \dfrac{Q.x_{i\mathrm{h}} - O.x_{i\mathrm{s}}}{O.x_{i\mathrm{e}} - O.x_{i\mathrm{s}}}(O.t_{\mathrm{e}} - O.t_{\mathrm{s}}) + O.t_{\mathrm{s}} & \text{if } O.x_{i\mathrm{e}} < O.x_{i\mathrm{s}} \\ O.t_{\mathrm{s}} & \text{if } O.x_{i\mathrm{s}} = O.x_{i\mathrm{e}} \in [Q.x_{i\mathrm{l}}, Q.x_{i\mathrm{h}}] \\ \perp & \text{otherwise} \end{cases} \tag{3}$$

$$t_{i,\mathrm{e}} = \begin{cases} \dfrac{Q.x_{i\mathrm{h}} - O.x_{i\mathrm{s}}}{O.x_{i\mathrm{e}} - O.x_{i\mathrm{s}}}(O.t_{\mathrm{e}} - O.t_{\mathrm{s}}) + O.t_{\mathrm{s}} & \text{if } O.x_{i\mathrm{s}} < O.x_{i\mathrm{e}} \\ \dfrac{Q.x_{i\mathrm{l}} - O.x_{i\mathrm{s}}}{O.x_{i\mathrm{e}} - O.x_{i\mathrm{s}}}(O.t_{\mathrm{e}} - O.t_{\mathrm{s}}) + O.t_{\mathrm{s}} & \text{if } O.x_{i\mathrm{e}} < O.x_{i\mathrm{s}} \\ O.t_{\mathrm{e}} & \text{if } O.x_{i\mathrm{s}} = O.x_{i\mathrm{e}} \in [Q.x_{i\mathrm{l}}, Q.x_{i\mathrm{h}}] \\ \perp & \text{otherwise} \end{cases} \tag{4}$$

Temporal kNN Queries: A temporal kNN query is defined by a spatial range $([Q.x_{i\mathrm{l}}, Q.x_{i\mathrm{h}}])$ that all answers must satisfy, and a temporal value $(Q.t)$ where all answers are returned based on the difference between $Q.t$ and the time the object is in the spatial region. A temporal kNN query in the native space can be processed as follows.

Similar to the range query, for each node that the search algorithm encounters, we check if there is an intersection between the query's spatial bounding box and the R-tree nodes' spatial bounding box as in (5).

$$\langle R.\overline{x}_1, \ldots, R.\overline{x}_d \rangle \between \langle Q.\overline{x}_1, \ldots, Q.\overline{x}_d \rangle \tag{5}$$

If (5) is false, we ignore that node. Otherwise, we compute how close the node's time interval is to the query's time so that we know the order that we want to visit that node. We measure the closeness with t_{visit} which is computed as

follows.

$$t_{\text{visit}} = \begin{cases} R.t_{\text{l}} - Q.t & \text{if } Q.t < R.t_{\text{l}} \\ Q.t - R.t_{\text{h}} & \text{if } Q.t > R.t_{\text{h}} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

Nodes are visited in ascending order of t_{visit}. For this purpose, similar to [11], a priority queue of nodes sorted based on their t_{visit} is used. That is, for each node encountered, if its corresponding bounding box overlaps spatially with the query's spatial range, we compute its t_{visit} and insert this t_{visit} along with the node id in the priority queue. Then, we pop a node from the priority queue, load it from the disk, and examine its children's bounding box. This process continues until an object is popped from the queue and returned as the nearest answer so far. The user may request for the next nearest answer by continuing popping a node from the queue and examine bounding boxes of the children of the node until it pops an object id. Leaf nodes are handled differently from internal nodes, since we use the segment representation optimization instead of the bounding box, as we described earlier in the context of range search. We compute the time interval that the motion overlaps with the query's bounding box ($T_{\text{Q,O}}$) as follows.

$$T_{\text{Q,O}} = \cap_{i=1}^{d} T_{\text{Q,O},i} \quad \text{where} \quad T_{\text{Q,O},i} = [t_{i,\text{s}}, t_{i,\text{e}}] \cap [O.t_{\text{s}}, O.t_{\text{e}}] \quad (7)$$

We compute $t_{i,\text{s}}$ and $t_{i,\text{e}}$ just the same as (3) and (4). If $T_{\text{Q,O}}$ results in $\varnothing$, the object is ignored. Otherwise, we compute t_{visit} for the object as follows.

$$t_{\text{visit}} = \begin{cases} O.t_{\text{s}} - Q.t & \text{if } Q.t < O.t_{\text{s}} \\ Q.t - O.t_{\text{e}} & \text{if } Q.t > O.t_{\text{e}} \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

This t_{visit} is then inserted to the priority queue along with the object id. Note that, in case of a one-sided kNN query – e.g., the most recent or the nearest future, (6) is still applicable except that we will not visit the node (or the object) for which $Q.t < R.t_{\text{l}}$ (or $Q.t < O.t_{\text{s}}$) in case of a "most recent" query. Similarity, we will not visit the node (or the object) for which $Q.t > R.t_{\text{h}}$ (or $Q.t > O.t_{\text{e}}$) in case of a "nearest future" query.

Spatial kNN Queries: Recall that a spatial kNN query is defined by a temporal range ($[Q.t_{\text{l}}, Q.t_{\text{h}}]$), where all answers must satisfy, and a spatial location ($Q.x_i$) where all answers are returned based on how close its location ($O.x_i$) is to $Q.x_i$. The search algorithm for a spatial kNN query starts from the root node of the tree and visit nodes whose bounding box R satisfies the condition: $R.\bar{t} \between Q.\bar{t}$. If this condition is false, the node is ignored. Otherwise, we calculate the order that each node should be visited. We visit each node based on the ascending

order of s_{visit} which is computed as follows:

$$s_{\text{visit}} = \sqrt{\sum_{i=0}^{d}(P.x_i - Q.x_i)^2} \quad \text{where} \quad P.x_i = \begin{cases} R.x_{i\text{l}} & \text{if } R.x_{i\text{l}} > Q.x_i \\ R.x_{i\text{h}} & \text{if } R.x_{i\text{h}} < Q.x_i \\ Q.x_i & \text{otherwise} \end{cases} \tag{9}$$

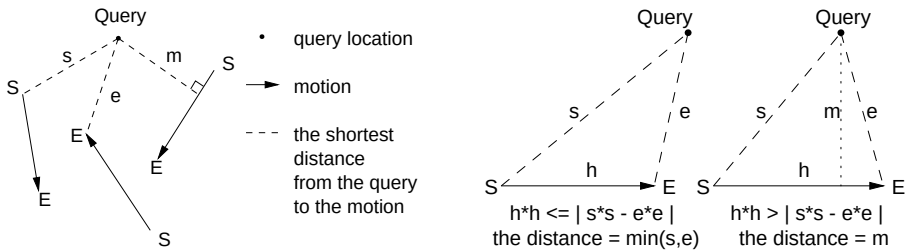
That is, s_{visit} is the shortest distance from a box to the query point. For the leaf nodes where the actual motion are stored, we compute the shortest distance (s_{visit}) from the line segment corresponding to the motion of the object to the query point as follows. First, we calculate the time interval ($[t_{\text{s}}, t_{\text{e}}]$) that the motion intersect the query along the temporal axis. Next we interpolate the location of object at time t_{s} and t_{e} as S and E respectively. That is,

$$[t_{\text{s}}, t_{\text{e}}] = [O.t_{\text{s}}, O.t_{\text{e}}] \cap [Q.t_{\text{s}}, Q.t_{\text{e}}] \tag{10}$$

$$S.x_i = O.x_{i\text{s}} + (O.x_{i\text{e}} - O.x_{i\text{s}})\frac{t_{\text{s}} - O.t_{\text{s}}}{O.t_{\text{e}} - O.t_{\text{s}}} \tag{11}$$

$$E.x_i = O.x_{i\text{s}} + (O.x_{i\text{e}} - O.x_{i\text{s}})\frac{t_{\text{e}} - O.t_{\text{s}}}{O.t_{\text{e}} - O.t_{\text{s}}} \tag{12}$$

Let s be the spatial distance between the query point and S; e be the spatial distance between the query point and E; and h be the spatial distance that the object travels from S to E. The shortest distance from the query to the object may be s, e, or neither (i.e., m) as shown in Fig. 3. Using Pythagoras' theorem



(a) the distance from a query to a motion

(b) identifying the shortest distance

Fig. 3. Computing the shortest distance from a query to each motion

the shortest distance can be computed as follows: if $h^2 \leq |s^2 - e^2|$, the shortest distance is $\min(s, e)$. Otherwise, the shortest distance is $m = \sqrt{e^2 - (\frac{h^2 - s^2 + e^2}{2h})^2}$ Once we get the shortest distance between the motion and the query point during the query's time interval, we insert this distance along with the motion id into the priority queue.

Indexing Other Motion Types: The NSI strategy can be extended in a straightforward fashion to support objects with motion other than linear translation by representing their motion trajectories using a bounding box that covers the trajectory regardless of its shape. However, as discussed earlier, the bounding box representation will result in false admissions. The problem of false positives can be alleviated in one of the following two ways. First, (similar to the approach discussed above for linear translation), object motion at the leaf level can be represented using a higher order curve (based on the location function associated with the motion). For each motion type we will have to develop techniques to determine temporal intercepts between the motion representation and the query along a spatial dimension (to evaluate spatio-temporal range query) and techniques to evaluate t_{visit} and s_{visit} to evaluate kNN queries.

An alternative approach to alleviate the false admission problem might be to use multiple smaller bounding boxes to cover the trajectory instead of a single large bounding box. The approach remains correct as long as the set of bounding boxes together cover the entire motion trajectory. Using multiple small boxes has an advantage that it reduces the number of false admissions since it represents more compact envelopes of the trajectory. Furthermore, it potentially reduces the overlap between the index nodes resulting in better performance. Nevertheless, it increases the size of the index which could result in the increase in the height of the tree causing increased I/O. Details of how motion trajectory can be segmented into bounding boxes and the resulting tradeoff can be found in [9].

3.3 Parametric Space Indexing Technique (PSI)

Unlike NSI strategy in which motion is represented in the native space, in the PSI strategy, motion is represented in the parametric space consisting of motion parameters. The PSI strategy works as follows. Let the motion parameters be $O.x_{1...\text{d}}$, $O.v_{1...\text{d}}$, and $[O.t_\text{s}, O.t_\text{e}]$ where x_i and v_i are the location and velocity of object O in the i^{th} spatial dimension and $[O.t_\text{s}, O.t_\text{e}]$ represents the valid time interval of the motion. In the PSI approach, an R-tree is used to index a $(2d + 1)$-dimensional parametric space where the dimensions correspond to $x_{1...\text{d}}$, $v_{1...\text{d}}$ and t. Each motion requires storage of $(2d + 2)$ floating-point values since each motion stores a temporal range $[t_\text{s}, t_\text{e}]$ that indicates the valid time interval of the motion. The motion representation is interpreted as a line segment in the parametric space parallel to the time axis connecting points $\langle O.x_{1...\text{d}}, O.v_{1...\text{d}}, O.t_\text{s}\rangle$ and $\langle O.x_{1...\text{d}}, O.v_{1...\text{d}}, O.t_\text{e}\rangle$. Each bounding box in the parametric space requires a storage of $2(2d+1)$ floating-point values corresponding to ranges in each of the parametric dimensions – $x_{1...\text{d}}$, $v_{1...\text{d}}$ and t.

Our parametric space representation differs from the ones in [5,12] in that we store the temporal range in which the motion is valid and we do not project the motion back to a global time reference. This overcomes the limitation of PSI strategies based on storing motion relative to the global time reference, as

we discussed in Sect. 2. An additional benefit of storing the temporal range explicitly is that now historical queries can easily be supported, provided that historical data are not deleted from the index.

While parametric representation of motion is straightforward, to evaluate a query either the condition specified in the query (which is a condition over the native space) needs to be transformed into an equivalent condition over the parametric space, or alternatively, the bounding box corresponding to a node in parametric space needs to be mapped to a corresponding region in the native space.

Spatio-Temporal Range Queries: Let $\langle Q.\overline{x}_1, ..., Q.\overline{x}_d, Q.\overline{t}\rangle$ be the range query. Given the parametric space representation of moving objects, we want to be able to answer spatio-temporal range queries efficiently. Identical to searching in the native space, the search over the parametric index begins at the root of the index structure and nodes in the index structure are recursively examined. Let R be a node in the tree and the parametric subspace corresponding to R be a bounding box $\langle \overline{x}_1, \ldots, \overline{x}_d, \overline{v}_1, \ldots, \overline{v}_d, \overline{t}\rangle$. Node R is visited by the search algorithm if there may exist an object O in the parametric subspace indexed by R such that the location of O during the time interval $\overline{t}$ satisfies the spatial constraint of query Q. That is,

$$\exists t \ \{t \in (Q.\overline{t} \cap R.\overline{t}) \wedge \langle R.X_1(t), ..., R.X_d(t)\rangle \in \langle Q.\overline{x}_1, ..., Q.\overline{x}_d\rangle\} \tag{13}$$

To check if there exists such an object O inside R, we need to compute the time interval that the bounding box query may overlap with any motion (projected in the native space) that has parameters within the range of R. We can calculate such a time interval along each spatial dimension and intersect them as follows.

$$T_{\mathrm{Q,R}} = \cap_{i=1}^{d} T_{\mathrm{Q,R},i} \tag{14}$$

where the time interval $T_{\mathrm{Q,R}}$ is the time interval that Q overlaps R and $T_{\mathrm{Q,R},i}$ is the time interval that Q overlaps R along dimension i. $T_{\mathrm{Q,R},i}$ can be calculated as $T_{\mathrm{Q,R},i} = [\min(t'), \max(t')]$ where t' satisfies the following equation.

$$Q.x_{i\mathrm{l}} \leq x_i + v_i \cdot (t' - t) \leq Q.x_{i\mathrm{h}} \tag{15}$$

subject to the following constraints: (1) $[t, x_i, v_i] \in R$ (the constraint of the node's bounding box), (2) $t' \in [R.t_{\mathrm{l}}, R.t_{\mathrm{h}}] \cap [Q.t_{\mathrm{l}}, Q.t_{\mathrm{h}}]$ (the answer's constraint), and (3) $t' \geq t$ which guarantees that the starting time of the motion, t, is before the time, t', that the motion intersects the query. Note that (15) is a non-linear equation with a set of linear constraints. There is no standard technique to solve such a non-linear equation. Fortunately, we can simply solve $T_{\mathrm{Q,R},i}$ by utilizing the specific properties of our problem domain (i.e., by mapping the parametric bounding box to a corresponding trapezoid region in the native space). There may be three cases of such mappings as shown in Fig. 4. In Fig. 4 (b) where

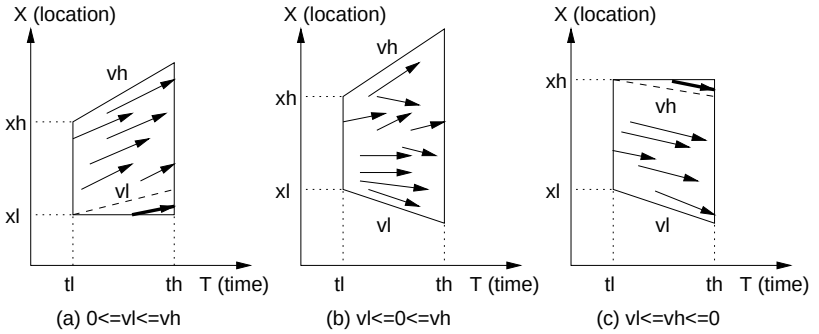


Fig. 4. A projection of a parametric bounding box onto the native space

$R.v_{i\mathrm{l}} \leq 0 \leq R.v_{i\mathrm{h}}$, the left and right borders of the trapezoid correspond to $R.t_{\mathrm{l}}$ and $R.t_{\mathrm{h}}$ respectively. The lower and upper bounds of the left border of the trapezoid correspond to $R.x_{i\mathrm{l}}$ and $R.x_{i\mathrm{h}}$ respectively. The slope of the lower and the upper borders of the trapezoid correspond to $R.v_{i\mathrm{l}}$ and $R.v_{i\mathrm{h}}$ respectively. That is, all motion in the bounding box starts moving from the range of location within $[R.x_{i\mathrm{l}}, R.x_{i\mathrm{h}}]$, moves with the range of velocity within $[R.v_{i\mathrm{l}}, R.v_{i\mathrm{h}}]$, and the starting time and the ending time of each motion are within the range of $[R.t_{\mathrm{l}}, R.t_{\mathrm{h}}]$. In Fig. 4 (a) where $0 \leq R.v_{i\mathrm{l}}$, unlike Fig. 4 (b), the slope of the lower border of the trapezoid is zero instead of $R.v_{i\mathrm{l}}$ since there may be some motion that starts at time t during $[R.t_{\mathrm{l}}, R.t_{\mathrm{h}}]$ from location $R.x_{i\mathrm{l}}$ (as shown with a thick arrow in the figure). Similarly, in Fig. 4 (c), the slope of the upper border of the trapezoid is zero instead of $R.v_{i\mathrm{h}}$. We use this trapezoid representation of a parametric bounding box in the native space to compute the overlap between the query and each R-tree node as follows. First, let's ignore the temporal range of the query and calculate when the trapezoid overlaps the spatial range of the query along each dimension. Let $[t_{i,\mathrm{s}}, t_{i,\mathrm{e}}]$ be the temporal range that the trapezoid overlaps the spatial range of query. Then, we calculate $T_{\mathrm{Q,R},i}$ as follows.

$$T_{\mathrm{Q,R},i} = [t_{i,\mathrm{s}}, t_{i,\mathrm{e}}] \cap [Q.t_{\mathrm{l}}, Q.t_{\mathrm{h}}] \cap [R.t_{\mathrm{l}}, R.t_{\mathrm{h}}] \tag{16}$$

$$\text{where} \quad t_{i,\mathrm{s}} = \begin{cases} R.t_{\mathrm{l}} + \dfrac{Q.x_{i\mathrm{l}} - R.x_{i\mathrm{h}}}{R.v_{i\mathrm{h}}} & \text{if } Q.x_{i\mathrm{l}} > R.x_{i\mathrm{h}} \wedge R.v_{i\mathrm{h}} > 0 \\ R.t_{\mathrm{l}} + \dfrac{Q.x_{i\mathrm{h}} - R.x_{i\mathrm{l}}}{R.v_{i\mathrm{l}}} & \text{if } Q.x_{i\mathrm{h}} < R.x_{i\mathrm{l}} \wedge R.v_{i\mathrm{l}} < 0 \\ R.t_{\mathrm{l}} & \text{if } Q.\overline{x}_i \between R.\overline{x}_i \\ \bot & \text{otherwise} \end{cases} \tag{17}$$

$$t_{i,\mathrm{e}} = \begin{cases} \bot & \text{if } t_{i,\mathrm{s}} = \bot \\ R.t_{\mathrm{h}} & \text{otherwise} \end{cases} \tag{18}$$

If $t_{i,\mathrm{s}} > t_{i,\mathrm{e}}$ or $t_{i,\mathrm{s}}$ and $t_{i,\mathrm{e}}$ are undefined ($\perp$), $T_{\mathrm{Q,R},i}$ is $\varnothing$. Like range queries in the native space index, if $T_{\mathrm{Q,R}} \neq \varnothing$, the search algorithm will visit R.

For a leaf node where motions $\langle \bar{t}, x_1, \ldots, x_d, v_1, \ldots, v_d \rangle$ are stored, we can calculate $t_{i,\mathrm{s}}$ and $t_{i,\mathrm{e}}$ as in (3) and (4) by replacing $\frac{O.x_{i\mathrm{e}} - O.x_{i\mathrm{s}}}{O.t_{\mathrm{e}} - O.t_{\mathrm{s}}}$ in (3) and (4) with $O.v_i$. That is,

$$t_{i,\mathrm{s}} = \begin{cases} O.t_{\mathrm{s}} + \dfrac{Q.x_{i\mathrm{l}} - O.x_i}{O.v_i} & \text{if } O.v_i > 0 \\ O.t_{\mathrm{s}} + \dfrac{Q.x_{i\mathrm{h}} - O.x_i}{O.v_i} & \text{if } O.v_i < 0 \\ O.t_{\mathrm{s}} & \text{if } O.v_i = 0 \wedge Q.x_{i\mathrm{l}} \leq O.x_i \leq Q.x_{i\mathrm{h}} \\ \perp & \text{otherwise} \end{cases} \tag{19}$$

$$t_{i,\mathrm{e}} = \begin{cases} O.t_{\mathrm{s}} + \dfrac{Q.x_{i\mathrm{h}} - O.x_i}{O.v_i} & \text{if } O.v_i > 0 \\ O.t_{\mathrm{s}} + \dfrac{Q.x_{i\mathrm{l}} - O.x_i}{O.v_i} & \text{if } O.v_i < 0 \\ O.t_{\mathrm{e}} & \text{if } O.v_i = 0 \wedge Q.x_{i\mathrm{l}} \leq O.x_i \leq Q.x_{i\mathrm{h}} \\ \perp & \text{otherwise} \end{cases} \tag{20}$$

Similar to the testing of trapezoid intersection with the query's bounding box, if $t_{i,\mathrm{s}} > t_{i,\mathrm{e}}$ or $t_{i,\mathrm{s}}$ and $t_{i,\mathrm{e}}$ are undefined ($\perp$), $T_{\mathrm{Q,R},i}$ is $\varnothing$ and thus $T_{\mathrm{Q,R}}$ is also $\varnothing$. If $T_{\mathrm{Q,R}}$ is not $\varnothing$, the motion is returned as an answer.

Temporal kNN Queries: Similar to the range query in the parametric space, we compute the temporal range ($T_{\mathrm{Q,R}}$) that the trapezoid corresponding to each node in parametric space (or the object motion stored in the leaf node) overlaps with the spatial range of the query by computing such a temporal range along each spatial dimension and intersecting them together ($T_{\mathrm{Q,R},i}$) as shown in (14). Similar to the range query computation in (16), we compute $T_{\mathrm{Q,R},i} = [t_{i,\mathrm{s}}, t_{i,\mathrm{e}}] \cap [R.t_{\mathrm{l}}, R.t_{\mathrm{h}}]$ where $t_{i,\mathrm{s}}$ and $t_{i,\mathrm{e}}$ is the same as (17) and (18). Since $T_{\mathrm{Q,R}}$ is the time interval that the query overlaps with each node or each motion, we want to visit each node and retrieve each motion in the ascending order of the time difference between $T_{\mathrm{Q,R}}$ and $Q.t$. Let t_{visit} be the time difference which is also the order that we will visit each node (or motion). Let $[t_{\mathrm{s}}, t_{\mathrm{e}}] = T_{\mathrm{Q,R}}$, we compute t_{visit} as follows.

$$t_{\mathrm{visit}} = \begin{cases} t_{\mathrm{s}} - Q.t & \text{if } t_{\mathrm{s}} > Q.t \\ Q.t - t_{\mathrm{e}} & \text{if } Q.t > t_{\mathrm{e}} \\ 0 & \text{otherwise.} \end{cases} \tag{21}$$

Spatial kNN Queries: Like spatial kNN queries in the native space, we check if the query overlaps with the node temporally. If not, we ignore that node. Let $[P.t_{\mathrm{l}}, P.t_{\mathrm{h}}] = [R.t_{\mathrm{l}}, R.t_{\mathrm{h}}] \cap [Q.t_{\mathrm{l}}, Q.t_{\mathrm{h}}]$ be the temporal overlap between the node and the query. If $[P.t_{\mathrm{l}}, P.t_{\mathrm{h}}] = \varnothing$, the search algorithm will not visit the node. Otherwise, we compute the shortest distance between the query's bounding

box and the node's boundary (s_{visit}). To compute s_{visit}, we first project the parametric bounding box during the time interval $[P.t_{\text{l}}, P.t_{\text{h}}]$. to a bounding box in the native space as follows. Let $\langle [P.t_{\text{l}}, P.t_{\text{h}}], [P.x_{i\text{l}}, P.x_{i\text{h}}] \rangle$ be such projection. We compute $P.x_{i\text{l}}$ and $P.x_{i\text{h}}$ as follows.

$$P.x_{i\text{l}} = \begin{cases} R.x_{i\text{l}} + R.v_{i\text{l}}(P.t_{\text{h}} - R.t_{\text{l}}) & \text{if } R.v_{i\text{l}} < 0 \\ R.x_{i\text{l}} & \text{otherwise} \end{cases} \tag{22}$$

$$P.x_{i\text{h}} = \begin{cases} R.x_{i\text{h}} + R.v_{i\text{h}}(P.t_{\text{h}} - R.t_{\text{l}}) & \text{if } R.v_{i\text{h}} > 0 \\ R.x_{i\text{h}} & \text{otherwise} \end{cases} \tag{23}$$

That is, we project the upper border of the parametric bounding box to the upper border of the native bounding box P and project the lower border of the parametric bounding box to the lower border of the native bounding box P. Then we compute the shortest distance (s_{visit}) between the query point $Q.x_i$ the bounding box P as follows.

$$s_{\text{visit}} = \sqrt{\sum_{i=0}^{d} (P.x_i - Q.x_i)^2} \quad \text{where} \quad P.x_i = \begin{cases} P.x_{i\text{l}} & \text{if } P.x_{i\text{l}} > Q.x_i \\ P.x_{i\text{h}} & \text{if } P.x_{i\text{h}} < Q.x_i \\ Q.x_i & \text{otherwise} \end{cases} \tag{24}$$

In case of the leaf node where motions are stored, we compute the shortest distance (s_{visit}) from motion O to query Q the same way that we do for the native space except that the native space motion is represented by the starting and ending point of motion while the parametric space motion is represented by the starting point and the velocity. Given the starting location and the time interval of the motion, the velocity can be derived from the ending of the motion and vice versa. Thus, the native space calculation for the leaf level node can be applied to the parametric space calculation.

4 Empirical Evaluation

In this section, we experimentally study the performance of NSI and PSI strategies. We compare their disk access performance in terms of the number of disk accesses per query as well as their CPU performance in terms of the number of distance computations.

Data: We randomly generate motion for 5000 mobile objects over a 2-dimensional 100-by-100 grid. Each mobile object updates its motion parameters (i.e., velocity and location) approximately every 1 time unit and data is collected over 100 time units. This results in a total of 502504 linear motion segments generated. To study the impact of speed of the mobile object on the indexing strategy, three different data sets are generated with a varying length of motion segment along each spatial dimension per time unit. Data is generated for segment lengths corresponding to 1 (slow speed), 2 (medium speed), and 4

(high speed) units. Unless otherwise stated, the experiments reported are based on index generated for slow speed (though query performance results are similar for data sets corresponding to higher speed of objects).

Index Buildup: The generated motion segments are indexed using an R-tree (based on both the PSI and NSI strategies). An R-tree page size is set to 4KB with a 50% fill-factor for all nodes. In the NSI R-tree, the fanouts are 145 and 127 for the internal/leaf node level respectively and the height of the resulting tree is 3. There are 83 internal nodes and 6806 leaf nodes. In the PSI R-tree, the corresponding fanouts are 92 and 127. The height of the tree is 4. This is because a parametric bounding box requires an additional dimension to store velocity for each spatial axis. There are 124 internal nodes and 6151 leaf nodes. Note that the number of leaf nodes in the native space R-tree is surprisingly slightly higher (even though it has a higher fanout) compared to the number of leaf nodes in the parametric space R-tree. This suggests that an addtional dimension per spatial axis in the parametric space index does not have much negative impact on the parametric space index.

Queries: To compare performance of NSI and PSI for different query types 1000 queries of each type are generated and results averaged for each experiment. To study performance of range query, randomly queries of different sizes – 0.25, 2.0, 4.0, 6.0, 8.0, and 10.0 along each spatial/teporal dimension – are generated. For temporal kNN queries, the temporal predicate is a single point while its spatial range predicate is set to a square range of size 6x6 and 10x10. For spatial kNN queries, the spatial predicate is a single point while its temporal range predicate is a random chosen range of sizes 6 and 10.

Experiments for Range Query: Fig. 5 plots the total number of disk accesses (including the fraction of disk access at the leaf level) for different query sizes. The figure shows that the number of page I/Os increases as the query size increases in both NSI and PSI. NSI outperforms PSI in terms of I/O performance. Each histogram bar in the plot is broken into two parts: the lower part represents the number of leaf-level disk accesses and the upper part the number of non-leaf disk accesses. As shown, in both NSI and PSI, the majority of disk accesses occurs at the leaf level.

Fig. 6 compares the CPU performance in terms of the number of distance computations. Similar to Fig. 5, each histogram bar shows the number of distance computations at the leaf- (lower part) and non-leaf part (upper part). The results show that NSI performs better than PSI in terms of CPU cost. Note that each distance computation of PSI is more costly than that of NSI. The number of leaf level distance computations is an indicator of how well the index structure organizes the data – lower the number of distance computations, better is the locality preserved by the index structure. A significantly higher number of leaf level distance computations of PSI compared to NSI indicates that NSI preserves locality better compared to PSI. This result is similar to that of [8], in which the PSI/NSI problem was studied in the context of indexing of objects with spatial extent as points in a parametric space.

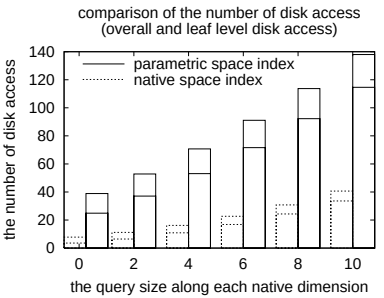


Fig. 5. I/O of range queries (NSI vs. PSI)

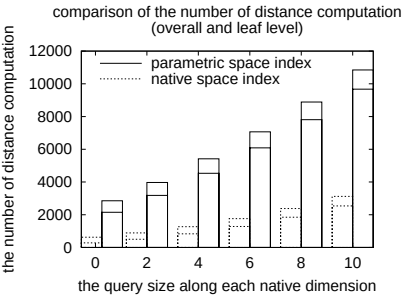


Fig. 6. CPU cost of range queries (NSI vs. PSI)

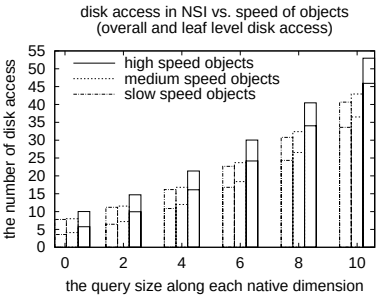


Fig. 7. I/O of range queries (NSI, varying motion speed)

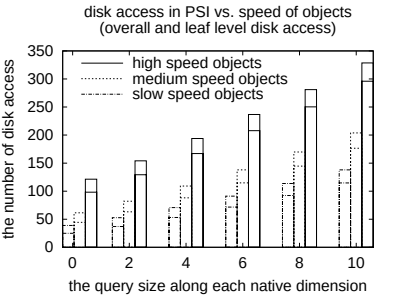


Fig. 8. I/O of range queries (PSI, varying motion speed)

Fig. 7 and Fig. 8 measure the impact of the speed of objects. Both figures indicate that query performance on high speed objects is worse than that of low speed objects. This is because high speed objects require larger boxes to cover. The figures also show that the impact of object's speed in PSI is higher than that in NSI. This is due to the fact that greater variations of speed, lead objects that are relatively close in parametric space to greatly varying locations in the actual, i.e., native space.

Experiments on Temporal kNN Query: Fig. 9 shows the relationship of the number of disk access and the number of temporal nearest neighbors for various spatial range of queries (i.e., 6x6 and 10x10). The result shows that, for each query size, the number of disk access increases fast at the begining when it starts exploring the tree and later the number of disk access increase slowly. A large query size require a higher number of disk access than a smaller query size in order to retrieve the same number of nearest neighbors. Similar to the performace of range queries, NSI appears to outperform PSI.

Experiments on Spatial kNN Query: Fig. 10 shows the relationship of the number of disk access and the number of spatial nearest neighbors for various

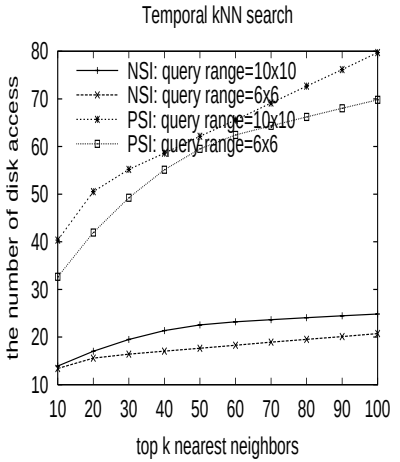


Fig. 9. I/O of temporal kNN (varying query spatial ranges in NSI and PSI)

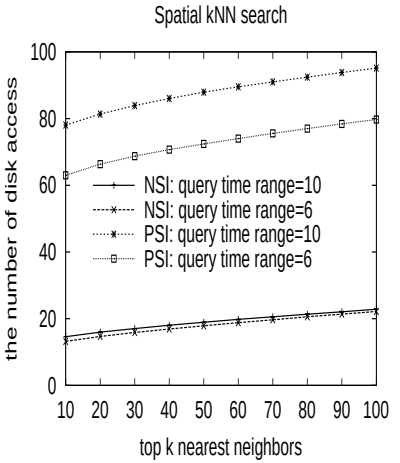


Fig. 10. I/O of spatial kNN (varying query temporal ranges in NSI and PSI

temporal range of queries (i.e., 6 and 10). The result is similar to the result of the spatial kNN queries.

5 Conclusion

This paper explores effective techniques for indexing and query processing over spatio-temporal databases storing mobile objects. The primary difficulty in developing query processing techniques arises from the dynamic nature of motion properties of objects – that is, the location of an object changes continuously as a function of time without an explicit update to the database. The paper explores algorithms to evaluate three different types of selection queries– namely, spatio-temporal range queries and spatial and temporal nearest neighbor queries.

The algorithms are developed under two two different representation and indexing techniques, native space- and parametric space- indexing (NSI/PSI). While NSI indexes motion in the original space in which it occurs, the PSI strategy indexes it in the space defined by the motion parameters. NSI preserves the locality of objects but also indexes "dead space" in the non-leaf levels of the index tree. PSI on the other hand uses the motion parameters to represent an object but suffers from the fact that objects that are close in parametric space might be somewhat divergent in real space and that objects that are far away in parametric space (e.g., two people starting at opposing sides of a room, moving with opposing velocities) may be at some time very close to each other (meeting in the middle of the room). Additionally, more complex types of motion (e.g., with acceleration) cannot be easily represented. In our future work we intend

to establish in more detail, both analytically and experimentally the conditions under which NSI/PSI should be used.

Acknowledgements. This work was supported in part by the National Science Foundation under Grant No. IIS-0086124 and in part by the Army Research Laboratory under Cooperative Agreements No. DAAL-01-96-2-0003 and No. DAAD-19-00-1-0188.

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 Int'l Conf. on Management of Data*, pages 322–331, May 1990.
3. A. Guttman. R-tree: a dynamic index structure for spatial searching. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 47–57, June 1984.
4. G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Int'l Symposium on Large Spatial Databases*, 1995.
5. G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Symposium on Principles of Database Systems*, 1999.
6. D. Lomet and B. Salzberg. The hB-Tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. on Database Systems*, 15(4):625–658, 1990.
7. D. Lomet and B. Salzberg. The performance of a multiversion access method. In *ACM SIGMOD Int'l Conf. on Management of Data*, 1990.
8. J. A. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 343–352, 1990.
9. K. Porkaew. Database support for similarity retrieval and querying mobile objects. Technical report, PhD thesis, University of Illinois at Urbana-Champaign, 2000.
10. J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *ACM SIGMOD Int'l Conf. on Management of Data*, 1981.
11. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *ACM SIGMOD Int'l Conf. on Management of Data*, 1995.
12. S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD Int'l Conf. on Management of Data*, 2000.
13. B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.
14. H. Samet. The quadtree and related hierarchial data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.
15. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+ tree: A dynamic index for multi-dimensional objects. In *Int'l Conf. on Very Large Data Bases*, 1987.
16. J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree based dynamic attribute indexing method. *Computer Journal*, 41(3):185–200, 1998.

17. Y. Theodoridis, T. Sellis, A. N. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *Int'l Conf. on Scientific and Statistical Database Management*, pages 123–132, 1998.
18. O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Int'l Conf. on Scientific and Statistical Database Management*, pages 111–122, 1998.