Processing Moving Queries over Moving Objects Using Motion-Adaptive Indexes

Buğra Gedik, *Student Member*, *IEEE Computer Society*, Kun-Lung Wu, *Senior Member*, *IEEE*, Philip S. Yu, *Fellow*, *IEEE*, and Ling Liu, *Senior Member*, *IEEE*

Abstract—This paper describes a *motion-adaptive* indexing scheme for efficient evaluation of moving continual queries (MCQs) over moving objects. It uses the concept of *motion-sensitive bounding boxes (MSBs)* to model moving objects and moving queries. These bounding boxes automatically adapt their sizes to the dynamic motion behaviors of individual objects. Instead of indexing frequently changing object positions, we index less frequently changing object and query *MSB*s, where updates to the bounding boxes are needed only when objects and queries move across the boundaries of their boxes. This helps decrease the number of updates to the indexes. More importantly, we use *predictive query results* to optimistically precalculate query results, decreasing the number of searches on the indexes. Motion-sensitive bounding boxes are used to incrementally update the predictive query results. Furthermore, we introduce the concepts of *guaranteed safe radius* and *optimistic safe radius* to extend our motion-adaptive indexing scheme to evaluating moving continual k-nearest neighbor (kNN) queries. Our experiments show that the proposed motion-adaptive indexing scheme is efficient for the evaluation of both moving continual range queries and moving continual kNN queries.

Index Terms—Moving object databases, spatio-temporal indexing, continual queries.

1 Introduction

With the continued advances in mobile computing and positioning technologies, such as GPS [16], location management has become an active area of research. Several research efforts have been made to address the problem of indexing moving objects or moving object trajectories to support efficient evaluation of continual spatial queries. Our focus in this paper is on *moving continual queries over moving objects* (MCQs for short). There are two major types of MCQs: *moving continual range queries* and *moving continual k-nearest neighbor queries*.

Efficient evaluation of MCQs is an important issue in both mobile systems and moving object tracking systems. Research on evaluating range queries over moving object positions has so far focused on static continual range queries [19], [11], [3]. A static continual range query specifies a spatial range together with a time interval and tracks the set of objects that locate within this spatial region over the given time period. The result of the query changes as the objects being queried move over time. Although similar, a moving continual range query exhibits some fundamental differences when compared to a static continual range query. A moving continual range query has an associated moving object, called the *focal object* of the query [7]; the spatial region of the query moves continuously as the query's focal object moves. Moving continual queries introduce a new challenge in indexing, due mainly to the highly dynamic nature of both queries and objects.

MCQs have different applications, such as environmental awareness, object tracking and monitoring, location-based services, virtual environments, and computer games, to name a few. Here is an example of a moving continual query, MCQ_1: "Give me the positions of those customers who are looking for taxis and are within five miles (of my location at each instant of time or at an interval of every minute) during the next 20 minutes," posted by a taxi driver on the road. The focal object of MCQ_1 is the taxi on the road. Another example is MCQ_2: "Give me the number of friendly units within a five-mile radius around me during the next two hours," posted by a soldier equipped with mobile devices marching in the field, or a moving tank in a military setting. The focal object of MCQ_2 is the soldier marching in the field or the moving tank.

Different specializations of MCQs can result in interesting classes of MCQs. One is called *moving continual queries over static objects*, where the target objects are stationary objects in the query region. An example of such a query is MCQ_3: "Give me the locations and names of the gas stations offering gasoline for less than $1.20 per gallon within 10 miles, during the next half an hour," posted by a driver of a moving car, where the focal object of the query is the car on the move and the target objects are the gas stations within 10 miles with respect to the location of the car. Another interesting specialization is the so called *static continual queries over moving objects*, where the queries are posed with static focal objects or without focal objects. An example query is MCQ_4: "Give me the list of AAA vehicles that are currently on service call in downtown Atlanta (or five miles from my office location), during the next hour." Note that these specializations of MCQs are computationally easier to evaluate. Our focus in this paper is the evaluation of MCQs in their most general form, such as MCQ_1 and MCQ_2.

Due to frequent updates to the index structures, traditional indexing approaches built on moving object positions generally do not work well for MCQs [19], [11]. In order to tackle this problem, several researchers have introduced alternative approaches based on the idea of indexing the

- *B. Gedik and L. Liu are with the College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332-0280. E-mail: {bgedik, lingliu}@cc.gatech.edu.*
- *K.-L. Wu and P.S. Yu are with the IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: {klwu, psyu}@us.ibm.com.*

Manuscript received 6 Jan. 2005; revised 7 Sept. 2005; accepted 25 Oct. 2005; published online 17 Mar. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0011-0105.

parameters of the motion functions of the moving objects [12], [20], [24], [1]. They effectively alleviate the problem of frequent updates to the indexes, as the indexes need to be updated only when the parameters change. These approaches are mostly based on R-tree-like structures and produce time-parameterized minimum bounding rectangles that enlarge continuously [20], [24], [19]. As a consequence of enlarged bounding rectangles, the search performance can deteriorate over time and the index structures may need to be reconstructed periodically [19], [20]. As far as update costs are concerned, approaches based on time-parameterized rectangles [20], [24] can provide excellent performance. However, they are not *sufficient* for processing MCQs. This is because they do not support incremental reevaluation of queries and the **continual** nature of these queries dictates that the same queries must be reevaluated at frequent intervals. Thus, there is a need for new methods that can evaluate these MCQs incrementally.

In this paper, we describe a *motion-adaptive indexing (MAI)* scheme for efficient processing of moving continual queries over moving objects. It uses the concept of *motion-sensitive bounding boxes (MSBs)* to model both moving objects and moving queries. Instead of indexing frequently changing object positions, we index less frequently changing object-and-query *MSB*s, where updates to the bounding boxes are needed only when objects and queries move across the boundaries of their boxes. This helps decrease the number of updates performed on the indexes. However, the main use of *MSB*s is to facilitate incremental processing of MCQs. We provide two techniques to reduce the costs of query reevaluation and search on the *MSB* indexes. First, we optimistically precalculate query results and incrementally maintain such *predictive query results* under the presence of object motion changes. *MSB*s are used to control the amount of precomputation to be performed for calculating the predictive query results and to decide when the results need to be updated. Second, we support *motion-adaptive* indexing. We automatically adapt the sizes of *MSB*s to the changing motion behaviors of the corresponding individual objects. By adapting to moving-object behavior at the granularity of individual objects, the moving queries can be evaluated faster by performing fewer IOs. Furthermore, we extend the *MAI* approach to the evaluation of moving continual *k-nearest neighbor* queries, by introducing the concepts of *guaranteed safe radius* and *optimistic safe radius* that are used to leverage the moving continual range queries for answering kNN queries.

Other interesting contributions of this paper are the development of an analytical model for estimating the cost of moving query evaluation and the use of analytical models to guide the setting and the adaptation of several system parameters for our proposed indexing scheme. The proposed motion-adaptive indexing scheme is independent of the underlying spatial index structures by design. In the experiments reported in this paper, we use both R*-trees and statically partitioned grids for measuring the performance of our indexing scheme. Our experimental results show that the motion-adaptive indexing scheme is efficient for the evaluation of both moving continual *range* queries and moving continual *k-nearest neighbor* queries. We report a series of experimental performance results for different workloads, including scenarios based on skewed object-and-query distribution, and demonstrate the effectiveness of our motion-adaptive indexing scheme through comparisons with other alternative indexing approaches.

The rest of the paper is organized as follows: We discuss the previous work in the literature related to querying and indexing moving object positions in Section 2. Section 3 gives an overview of the basic concepts and the system model. Section 4 describes the motion-adaptive indexing scheme for efficient evaluation of moving range queries. Section 5 extends the solution to the efficient evaluation of moving kNN queries. Section 6 reports various performance results to illustrate the effectiveness of the proposed approach. We conclude with a summary in Section 7.

2 Related Work

Research on moving object indexing can be broadly divided into two categories, based on 1) the current positions of the moving objects and 2) the trajectories of the moving objects. Our work belongs to the first category. An essential study dealing with the problem of indexing and querying moving object trajectories can be found in [18]. Continual queries are used as a useful tool for monitoring frequently changing information [25], [14]. In the spatial databases domain, continual queries are employed for continuously querying moving object positions. Most of the work on continual queries over moving object positions is either on static continual queries over moving objects [19], [11], [12], [3], [21], [29], [30] or on moving continual queries over static objects [23], [22]. None of the these works has addressed the problem of moving *continual* queries over moving objects.

In [19], velocity-constrained indexing and query indexing (Q-index) have been proposed for efficient evaluation of static continual range queries. The same problem is studied in [11]; however, the focus is on in-memory structures and algorithms. In [20], TPR-tree, an R-tree-based indexing structure, is proposed for indexing the motion parameters of moving objects by using time-parameterized rectangles and answering queries using this index. TPR*-tree, an extension of TPR-tree-optimized for queries that look into the future (predictive), is described in [24]. Note that even though TPR-related indexes [20], [24] support moving queries, these moving queries are predefined regions in the spatio-temporal domain. They are not the moving continual queries, such as MCQ_1 and MCQ_2, discussed in this paper. Recently, newer indexing schemes that improve upon the performance of TPR-trees have been introduced, such as STRIPES [17] and the B$^+$-tree-based indexing technique of [10]. Nevertheless, the focus of these works is on developing search-and-update-efficient indexing structures for managing moving object locations and they do not have special mechanisms to support continual queries, whereas our focus is on developing a logical indexing scheme that leverages already existing indexing structures to support efficient processing of MCQs through incremental evaluation. Advanced indexing structures can be integrated into our *MAI* approach by replacing the R*-tree-based object-and-query indexes we employ.

In [2], efficient query evaluation techniques for nearest-neighbor ($k = 1$) and reverse-nearest-neighbor queries are developed for moving queries over moving objects. CNN [23] gives an algorithm for precalculating k-nearest neighbors with a line segment representing the continuous motion of the query; however, the target objects are assumed to be static. In [32], techniques based on object-only indexing and query-only indexing are proposed to evaluate moving continuous kNN queries over moving objects. However, the solution is exclusive to kNN queries. In contrast, our approach supports range and kNN queries

TABLE 1
Comparison of Motion-Adaptive Index with Existing Approaches

	Query Types			System Properties				
	Moving Query Static Object	Static Query Moving Object	Moving Query Moving Object	Incremental Evaluation	Predictive Query Results	Index Independence	Motion Modeling	Motion Adaptation
MAI	•	•	•	•	•	•	•	•
SINA [15]	•	•	•	•		•		
TPR [20]	○ [1]	•	○ [1]				•	
DQ [13]	•	•	•	•			•	
CNN [23]	•				○ [2]		•	
Q-index [19]		•		•		•		

1. TPR-tree only supports moving queries with predefined paths.
2. CNN has per-result time intervals, not per-object.

within the same framework and uses object-and-query indexing at the same time to optimize the performance for a large range of parameters that include cases where object-only indexing falls short, as well as cases where query-only indexing is ineffective.

The concept of moving continual queries is to some extent similar to Dynamic Queries (DQ) [13]. A dynamic query is defined as a temporally ordered set of snapshot queries in [13]. This is a low-level definition as opposed to our definition of moving continual queries, which is more declarative and is defined from the users' perspective. The work done in [13] indexes the trajectories of the moving objects and describes how to efficiently evaluate dynamic queries that represent predictable or unpredictable movement of an observer. They also describe how new trajectories can be added when a dynamic query is actively running. Their assumptions are in line with their motivating scenario, which is to support rendering of objects in virtual-tour-like applications. Our work focuses on real-time evaluation of moving queries in real-world settings, where the trajectories of the moving objects are unpredictable and the queries can potentially be associated with moving objects inside the system. An important feature of our approach is its motion adaptiveness, allowing the query evaluation to be optimized according to the dynamic motion behavior of the objects. Our experiments have shown that such motion-adaptive capability offers significant performance gain for evaluating moving queries over moving objects.

The most relevant work to ours, in terms of its support for various types of continual spatial queries discussed in Section 1 and its ability to perform incremental evaluation, is the SINA [15] (and its kNN extension SEA-KNN [31]) algorithm that has been developed concurrently and independently with our work [8]. SINA employs hash-based indexing techniques for both objects and queries and generates positive and negative updates (incrementally) through a three-step process consisting of hashing, invalidation, and joining. However, there is an inherent difference between our approach and SINA. Specifically, motion modeling (described in Section 3.2) is integrated into our approach, which enables predictive query results and helps increase the system scalability by reducing the number of location updates received from the moving objects. It has been shown in [4] that the use of linear functions for motion modeling reduces the amount of updates to one-third in comparison to constant functions, for realistic thresholds. However, SINA works on raw location updates in the form of (x, y) coordinate pairs and is not designed to take advantage of motion modeling. On the other hand, motion modeling may introduce additional processing requirements on the moving objects. Fortunately, dead-reckoning algorithms for linear motion modeling are simple and can be implemented easily with cheap hardware or software. Besides these, the SINA approach is not motion-adaptive like our MAI approach, i.e., it does not optimize the system based on the movement characteristics of the individual objects. In summary, SINA and MAI are different in their assumptions and requirements with respect to the supports required by the mobile objects, as well as in terms of the specific techniques they employ for the purpose of query evaluation. However, both are intended to solve the same high-level problem of evaluating moving continuous queries over moving objects.

In [21], a two-level architecture is proposed, where there exist location preprocessors between the moving objects and the database. The location updates are propagated to the database only when the objects cross boundaries of their hash buckets, which are fixed. The database is aware of only the hash buckets and does not know exact positions of objects within the buckets. Some queries have to be propagated to location preprocessors that have the exact information. Going further in this direction, in [3], [7], and [9], two-level architectures that push the location filtering to mobile units were described.

Table 1 summarizes the comparison of our *MAI* approach with some of the existing approaches. Our approach is the most universal in handling various types of continual queries and has many desirable system properties, such as incremental evaluation of queries and motion adaptation.

3 The System Model

The basic elements of our system model are a set of moving or stationary objects and a set of moving or static continual (range or kNN) queries. A fundamental challenge we address in this paper is to study what kind of indexing scheme can efficiently answer the moving queries. Fast evaluation is critical for processing moving queries as it not only improves the freshness of the query results by enabling more frequent reevaluation, but also increases the scalability of the system by enabling timely evaluation of a large number of moving queries over a large number of moving objects.

3.1 Basic Concepts and Problem Statement

We denote the set of moving or stationary objects as O, where $O = O_m \cup O_s$ and $O_m \cap O_s = \emptyset$. O_m denotes the set of moving objects and O_s denotes the set of stationary objects. We denote the set of moving or static queries as Q, where $Q = Q_m \cup Q_s$ and $Q_m \cap Q_s = \emptyset$. Q_m denotes the set of moving continual range queries and Q_s denotes the set of static continual range queries. Since we focus on moving continual queries in this paper, from now on we use moving queries and moving continual queries interchangeably.

Moving Objects. We describe a moving object $o_m \in O_m$ by a quadruple: $\langle i_o, \bar{p}, \bar{v}, a_p \rangle$. Here, i_o is the unique object identifier, $\bar{p} = (p_x, p_y)$ is the current position of the moving object where p_x is its position in the x-dimension and p_y is its position in the y-dimension, $\bar{v} = (v_x, v_y)$ is the current velocity vector of the object, and a_p is a set of properties about the object. A stationary object can be modeled as a special case of moving object where the velocity vector is set to zero, $\forall o_s \in O_s, o_s.\bar{v} = (0, 0)$.

Moving Queries. We describe a moving query $q_m \in Q_m$ by a quadruple: $\langle i_q, i_o, r, f \rangle$. Here, i_q is the unique query identifier, i_o is the object identifier of the focal object of the query, r defines the shape of the spatial query region bound to the focal object of the query, and f is a Boolean predicate, called *filter*, defined over the properties (a_p) of the target objects of the query. Note that, r can be described by a closed shape description such as a rectangle or a circle. This closed shape description also specifies a binding point, through which it is bound to the focal object of the query. In the rest of the paper, we assume that a moving continual query specifies a circle as its range with its center serving as the binding point and we use r to denote the radius of the circle. A static spatial continual range query can be described as a special case where the query either has no focal object or the focal object is a stationary object. Namely, $\forall q_s \in Q_s, q_s.i_o = null \vee q_s.i_o \in O_s$. We assume that a static continual range query specifies a rectangle or a circle as its range.

Before we give an overview of our approach, we first review three basic types of indexing techniques for evaluating moving range queries over moving objects and discuss their advantages and inherent weaknesses.

Object-Only Indexing (*OI*). In the object-only indexing approach, a spatial index is built on the object positions. Each time a new object position is received, the object index is updated. At each query evaluation phase, all queries are evaluated against the object index. An inherent drawback of the basic object-only indexing approach is the reevaluation of all queries against the object index regardless of whether or not the object position changes are of interest to the query. Object-only indexing is open to optimizations that can decrease the number or cost of the updates on the object index (see velocity-constrained indexing in [19] and TPR-trees in [20]).

Query-Only Indexing (*QI*). In the query-only indexing approach, a spatial index is built on the spatial regions of the queries. Each time a new query position (the position of the query's focal object) is received, the query index is updated. At each query evaluation phase, each object position is evaluated against the query index and the queries that contain the object's position are determined. Note that this has to be done for every object as opposed to doing it only for objects that have moved since the last query evaluation phase. This is due to the fact that underlying queries are potentially moving. This significantly decreases the effectiveness of query-only indexing approach, although in the context of static continual range queries it has been shown that a query index may improve performance significantly [19], [29], [30].

Object-and-Query Indexing (*OQI*). In the object-and-query indexing approach, two spatial indexes are built, one for the object positions and another for the spatial regions of the queries. Each time an object position is received, the object index is updated. Similarly, each time a new query position (the position of a query's focal object) is received, the query index is updated. At each query evaluation phase, each *new* object position is evaluated against the query index and the queries that contain the object's position are determined. Then, the query results are updated differentially. Similarly, at each query evaluation phase, each *new* query position is evaluated against the object index and the new result of the query is determined. The OQI approach evaluates object positions against the query index only for those objects that have changed their positions since the last query evaluation phase, as opposed to all the objects required by the query-only indexing approach. The OQI approach also evaluates queries against the object index only for those queries that have moved since the last query evaluation phase, as opposed to all the queries required by the object-only indexing approach. Although the OQI approach incurs a higher cost due to the maintenance of an additional index structure, it is open to a wider range of optimizations to reduce the cost and it does not have certain restrictions of the object-only indexing or query-only indexing approach.

3.2 Overview of the Proposed Solution

Cognizant of the pros and cons of the above three basic indexing schemes, we propose a motion-adaptive indexing scheme for efficient processing of moving queries over moving objects. We use the concept of *motion-sensitive bounding boxes* to model the dynamic behavior of both moving objects and moving queries. Such bounding boxes are not updated unless the position of a moving object or the spatial region of a moving query exceeds the borders of its bounding box. Instead of indexing frequently changing object positions or spatial regions of moving queries, we index less frequently changing motion-sensitive bounding boxes. This significantly decreases the number of update operations performed on the indexes. Our indexing scheme maintains both an index of object-based motion-sensitive bounding boxes (denoted as $Index_o^{msb}$) and an index of query-based motion-sensitive bounding boxes (denoted as $Index_q^{msb}$).

More importantly, to address the problem of increased search cost due to frequent evaluation of queries, we employ two optimization techniques: 1) *predictive query results* and 2) *motion-adaptive indexing*. Query results are optimistically precomputed in the presence of object motion changes, with the amount of precomputation to be performed controlled by the motion-sensitive bounding boxes. The sizes of the motion-sensitive bounding boxes are dynamically adapted to the changing motion behaviors at the granularity of individual objects, allowing moving queries to be evaluated faster by performing fewer IOs. Fig. 1 gives a roadmap of methods applied for MCQ evaluation.

In the rest of this section, we describe the motion-modeling and motion update generation, which provides the foundation for *motion-sensitive bounding boxes* and *predictive query results*.

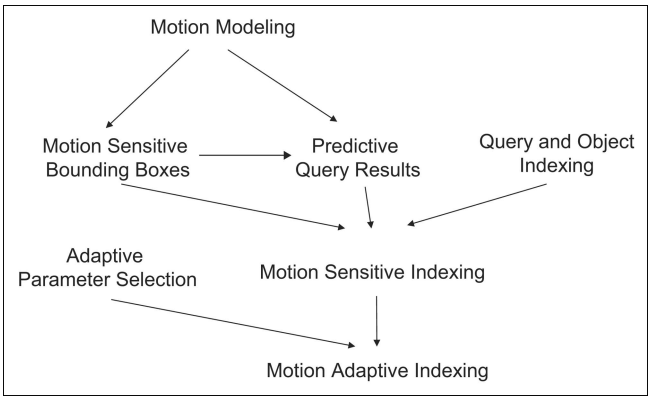


Fig. 1. Roadmap of methods applied for moving query evaluation.

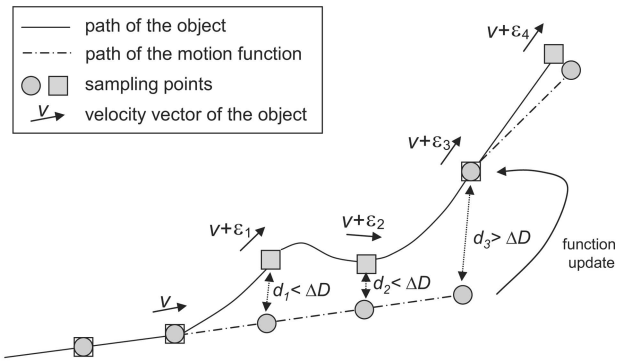


Fig. 2. Motion-update generation.

Motion Modeling. Modeling motions of moving objects for predicting their positions is a commonly used method in moving object indexing [27], [12]. In reality, a moving object moves and changes its velocity vector continuously. Motion modeling uses approximation for prediction. Concretely, instead of reporting their position updates each time they move, moving objects report their velocity vector and position updates only when their velocity vectors change and this change is significant enough. (This technique is known as dead reckoning [5].) In order to evaluate moving queries between the last update reporting and the next update reporting, the positions of the moving objects are predicted using a simple linear function of time. Given that the last received velocity vector of an object is $\bar{v}$, its position is $\bar{p}$ and the time its velocity update was recorded is t, the future position of the object at time $t + \Delta t$ can be predicted as $\bar{p} + \Delta t * \bar{v}$. We use a linear motion function in this paper, since it is the commonly used model in moving object databases [28]. We refer readers to [1] for a study of nonlinear motion modeling for moving object indexing.

Prediction-based motion modeling decreases the amount of information sent to the query-processing engine by reducing the frequency of position reporting from each moving object. Furthermore, it allows the system to optimistically precompute future query results. Below, we briefly describe how the moving objects generate and send their motion updates to the server, where the query evaluation is performed.

Motion Update Generation. In order for the moving objects to decide when to report their velocity vector and position updates, they need to periodically compute if their velocity vectors have changed significantly. Concretely, at each time step, a moving object samples its current position and calculates the difference between its current position and the position predicted by the dead-reckoning algorithm based on the last motion update it reported to the server. In case this difference is larger than a specified threshold, say, ΔD, the new motion function parameters are relayed to the server. Fig. 2 provides an illustration. The path of a moving object is depicted with a solid line, where its path predicted by the server is depicted with a dashed line. The small squares on the solid line represent the current positions sampled by the moving object at each time step, and the small circles on the dashed line represent the positions the server predicts the object to be at in each of the corresponding time steps.

4 Efficient Evaluation of Moving Continual Range Queries

In this section, we describe the motion-adaptive indexing scheme for efficient processing of moving range queries over moving objects. We first describe the concept of motion-sensitive bounding boxes, and then discuss the mechanisms used for computing predictive query results, and outline the motion-adaptive approach for determining the sizes of motion sensitive bounding boxes. In addition, we provide an overview of the algorithms used for creating and maintaining the motion-adaptive indexes, an analytical model for IO estimation, and the concrete mechanism that adaptively determines the bounding-box sizes based on the dynamically changing motion behaviors of moving objects and moving queries.

4.1 Motion-Sensitive Bounding Boxes

Motion-sensitive bounding boxes (*MSB*s) can be defined for both moving queries and moving objects. Given a moving object o_m, its associated *MSB* is calculated by extending the position of the object along each dimension by $\alpha(o_m)$ times the velocity of the object in that direction. Given a moving query q_m, the *MSB* of the moving query is calculated by extending the minimum bounding box of the query along each dimension by $\beta(q_m)$ times the velocity of the focal object of the query in that direction (see Fig. 3 for illustrations).

Let $Rect(l, m)$ denote a rectangle with l and m as any two endpoints of the rectangle that are on the same diagonal. Let $sign(\bar{x})$ denote a function over a vector $\bar{x}$, which replaces each entry in $\bar{x}$ with 1 if it is greater than or equal to 0, with -1 otherwise. Then, we define the *MSB* for a moving

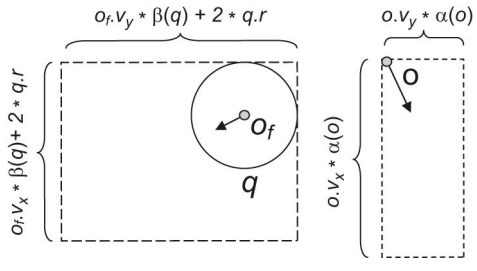


Fig. 3. Motion sensitive bounding boxes (*MSB*s).

technique	Overall Update Cost	Overall Search Cost
MSBs	↓ (due to less frequent updates to the indexes)	↑ (due to increased overlap in indexes)
Predictive Query Results (using MSBs)	–	↓ (due to incremental query evaluation)
Together	+ ↓	+ ↓

Fig. 4. Impact of *MSB*s and predictive query results on query evaluation cost.

object o and the *MSB* for a moving query q with focal object o_f as follows:

$$\forall o \in O_m, MSB(o) = Rect(o.pos, o.pos + \alpha(o) * o.vel)$$
$$\forall q \in Q_m, MSB(q) = Rect(o_f.pos - q.radius * w_s, o_f.pos + \beta(q) * q.vel + q.radius * w_s),$$

where w_s denotes the sign function $sign(q.vel)$.

For each moving query, its *MSB* is calculated and used in place of the query's spatial region in the query-based *MSB* index, that is $Index_q^{msb}$. Similarly, for each moving object, its *MSB* is calculated and used in place of the object's position in the object-based *MSB* index, that is, $Index_o^{msb}$.

An important feature of indexing motion-sensitive boxes of moving objects and moving queries is the fact that an *MSB* is not updated unless the query's spatial region or the object's position exceeds the borders of its motion-sensitive bounding box. When this happens, we need to invalidate the *MSB*. As a result, a new *MSB* is calculated and the $Index_q^{msb}$ or the $Index_o^{msb}$ is updated. This approach reduces the number of update operations performed on the spatial indexes and, thus, decreases the overall cost of updating the spatial indexes ($Index_o^{msb}$ and $Index_q^{msb}$). It is also crucial to note that, using *MSB*s does not introduce any inaccuracy in the query results, because *we store the motion function of the object or the query together with its* MSB *inside the spatial index*.

Although maintaining *MSB* indexes increase the cost of searching the index due to higher overlap of spatial objects being indexed, for appropriate values of the α and β parameters, the overall gain in the search cost due to the use of *MSB*s is significant, thanks to the incremental processing capabilities *MSB*s provide in conjunction with predictive query results. Concretely, when a query has not invalidated its *MSB* and has not changed its velocity vector, then the predictive results of the query are valid with regard to the objects for which no *MSB* invalidations or velocity vector changes have taken place. In case some of the objects had *MSB* invalidations or velocity vector changes, queries are not completely reevaluated. A query is completely reevaluated only when it has invalidated its *MSB* or it has changed its velocity vector. We will discuss the details of query evaluation in greater depth in Section 4.3. In summary, the incremental processing of queries helps minimize the overall search cost and compensates for the small increase in the per operation index search cost due to the use of MSBs. Fig. 4 summarizes the impact of using MSBs on the query evaluation in terms of update and search cost.

Furthermore, MSBs provide the following three advantages:

1. As opposed to approaches that alter the implementation of traditional spatial indexes for decreasing the update cost (like TPR-tree [20] or VCI index [19]), motion-sensitive bounding boxes require almost no significant change to the underlying spatial index implementation.
2. They form a basis for deciding for which objects to precalculate query results with respect to a query (see Section 4.3).
3. By performing size adaptation at the granularity of individual objects, they lead to significant reductions in IO cost (see Section 4.4).

In order to fully utilize the advantages made possible by *MSB*s in terms of query evaluation cost, we need mechanisms for dynamically determining the most appropriate values of the α and β parameters based on the motion behavior of moving objects and moving queries.

4.2 Predictive Query Results on a Per-Object Basis

It is well-known that one way of reducing IO and improving efficiency of evaluating moving queries is to precalculate future results of the continual queries. This approach has been used successfully in the context of continual moving kNN queries over *static* objects [23]. Most existing approaches to precalculating query results associate a time interval with each query that specifies the valid time for the precalculated results. One problem with per-query-based prediction in the context of moving queries over moving objects is the fact that a change on the motion function of any of the moving objects may cause the invalidation of some of the precalculated results. This motivates us to introduce *predictive query results*, where the prediction is conducted on per-object basis.

Given a query, its predictive query result differs from a regular query result in the sense that each object in the predictive query result has an associated time interval indicating the time period in which the object is *expected* to be included in the query result. We denote the predictive query result of query $q \in Q$ by $PQR(q)$. Each entry in a predictive query result takes the form $\langle o, [t_s, t_e] \rangle$. We call the entry associated with object $o \in O$ in $PQR(q)$ the *predictive query result entry* of object o with regard to query q, and the interval $[t_s, t_e]$ associated with object o the *valid prediction time interval* of the predictive query result entry.

Calculating the valid prediction time intervals is done as follows: Given a static continual range query and a moving object with its motion function, it is straightforward to calculate the intersection points of the query's spatial region and the ray formed by the moving object's trajectory (See case I in Fig. 5). Similarly, to calculate the intersection point of a moving query and a moving or nonmoving object (assuming that we only consider moving queries with circle- shaped spatial regions), we need to solve a quadratic function of time. Formally, let $q \in Q$ be a query with focal object $o_f \in O_m$, let $o \in O$ be an object, and let $Dist(a, b)$ denote the Euclidean distance between the two points a and b. We can calculate the time interval in which the object o is expected to be in the result set of query q by solving the formula:

$$Dist(o_f.\bar{p} + t * o_f.\bar{v}, o.\bar{p} + t * o.\bar{v}) \leq q_m.r.$$

Fig. 5 illustrates three different cases that arise in the calculation of the prediction-time interval for each per-object-based predictive query result entry.

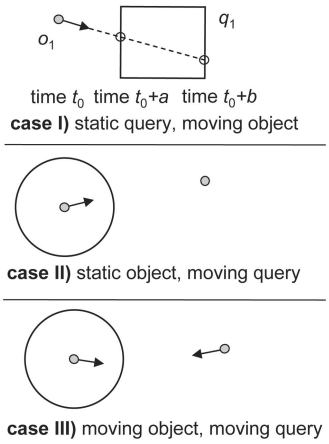


Fig. 5. Calculating intervals.

The predictive query results are precalculated on a per-object basis and the result entries are correct unless the motion function of the focal object of a query or the motion function of the moving object associated with the query result entry have changed within the valid prediction-time interval. As a result, there are two key questions to answer in order to effectively use the predictive query results in evaluating MCQs:

Prediction: *For each moving query, should we perform prediction on all moving objects? If not, how do we determine for which objects we should do prediction?*

Obviously, we should not perform prediction for objects that are far away from the spatial region of the query within a period of time, as the predicted results are less likely to hold until those objects reach to the proximity of the query.

Invalidation: *When and how to update the predictive results?*

This can be referred to as the invalidation policy for per-object-based prediction. The predictive query results may be invalid and, thus, need to be updated when the motion function of a moving query or the motion function of a moving object changes. In addition, the predictive results may need to be refreshed when the objects in the predictive query results have moved away from the proximity of the query or when the objects that did not participate in the prediction have entered the proximity of the query.

4.3 Determining Predictive Query Results Using MSBs

*MSB*s are used to effectively determine for which objects we should perform result prediction with respect to a query (answering the first question listed in Section 4.2). Concretely, for a given query, objects whose *MSB*s intersect with the query's *MSB* are considered as potential candidates of the query's predictive result. Fig. 6 gives an illustration of how predictive query results integrate with motion-sensitive bounding boxes. Consider the moving query q_1 with its query MSB and four moving objects $o_1, o_2, o_3,$ and o_4 as shown in Fig. 6. In the figure, o_1 is the focal object of query q_1 and the other three moving objects, $o_2, o_3,$ and o_4 are associated with their object *MSB*s. At time t_0, only objects o_2 and o_3 are subject to query q_1's *PQR*, as their *MSB*s intersect with the query's *MSB*. However, the valid prediction time interval of object o_3 with regard to query q_1 is empty because there is no such time interval during which o_3 is expected to be inside the query result of q_1. Thus, object o_3 should not be included in the *PQR* of query q_1. At some later time t_1, object o_2 and query q_1 remain inside their *MSB*s. However, objects o_3 and o_4 have changed their *MSB*s. As a result, objects o_2 and o_4 become potential candidates of query q_1's *PQR* at time t_1. Since o_2 has not changed its *MSB*, it remains included in q_1's *PQR*. By applying the valid prediction time interval test on o_4, we obtain a nonempty time interval with respect to q_1, during which o_4 is expected to be included in the query result. Thus, o_4 is added into the *PQR* of q_1.

In order to achieve an IO-efficient solution, the *MSB* sizes should be adjusted such that the *PQR*s are calculated for a sufficiently large set of objects to take advantage of precomputation. However, result prediction should not be performed for objects that are far away from a query, and, thus, are likely to invalidate their *PQR*s before becoming of interest to the query. We use α and β parameters to adjust the *MSB* sizes on a per-object/query basis to optimize this trade-off. The details are given in Section 4.5.

4.4 Motion-Adaptive Indexing

We have described the main ideas and mechanisms used in our motion-adaptive indexing scheme. In this section, we describe motion-adaptive indexing as a query evaluation technique that integrates the ideas and mechanisms presented so far for efficient processing of moving queries over moving objects.

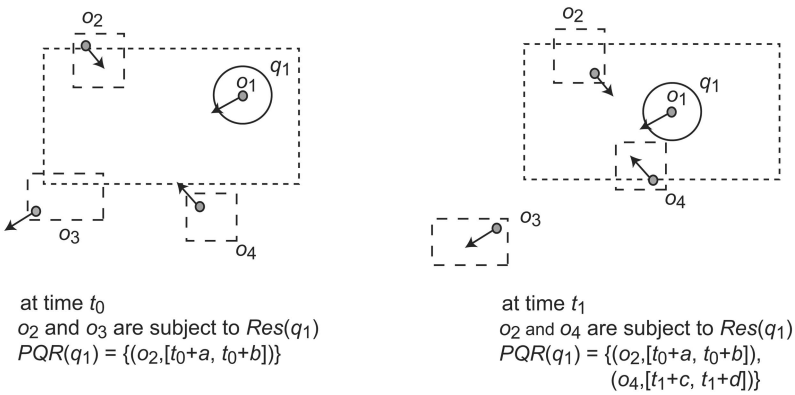


Fig. 6. An illustration of how *PQR*s integrate with *MSB*s.

Algorithm 1: Motion Update Received $u = \langle i_o, \overline{p}, \overline{v}, t \rangle$

$e_o = \langle i_o, i_q, \overline{p}, \overline{v}, t, B_{msb}, P_{cm}, V_{ch} \rangle \in MOT$
where $e_o.i_o = u.i_o$
$e_o.P_{cm} \leftarrow \gamma * (u.t - e_o.t) + (1 - \gamma) * e_o.P_{cm}$
$e_o.\overline{p} \leftarrow u.\overline{p}$; $e_o.\overline{v} \leftarrow u.\overline{v}$
$e_o.t \leftarrow u.t$; $e_o.V_{ch} \leftarrow$ **true**
if $e_o.i_q \neq$ **null then**
$e_q = \langle i_q, \overline{p}, \overline{v}, r, t, B_{msb}, P_{cm}, V_{ch} \rangle \in MQT$
where $e_q.i_q = e_o.i_q$
$e_q.P_{cm} = \gamma * (u.t - e_q.t) + (1 - \gamma) * e_q.P_{cm}$
$e_q.\overline{p} \leftarrow u.\overline{p}$; $e_q.\overline{v} \leftarrow u.\overline{v}$;
$e_q.t \leftarrow u.t$; $e_q.V_{ch} \leftarrow$ **true**
end if

Fig. 7. Motion update processing.

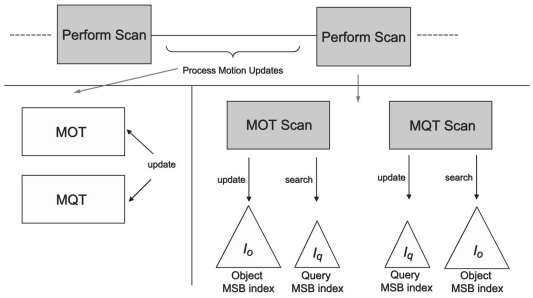


Fig. 8. General view of query evaluation.

4.4.1 Processing Moving Queries: An Overview

The evaluation of moving queries is performed through query evaluation phases executed periodically with regular time intervals of P_s (*scan period*) seconds. We build two spatial *MSB* indexes, $Index_o^{msb}$ for the objects and $Index_q^{msb}$ for the queries. $Index_o^{msb}$ stores *MSB*s of the objects accompanied by the associated motion functions as data. Static objects have point *MSB*s. Similarly, $Index_q^{msb}$ stores the *MSB*s of the queries accompanied by the associated motion functions of the focal objects of the queries and their radii as data. Static queries have *MSB*s equal to their minimum bounding rectangles and they do not have associated motion functions.

We create and maintain two tables, a moving object table and a moving query table. They store information regarding the moving objects and moving queries. The static queries and static objects are included in the spatial *MSB* indexes but not in the two tables. The periodic evaluation is performed by scanning these tables at each query evaluation phase and performing updates and searches on the spatial indexes as needed in order to incrementally maintain the query results as objects and the spatial regions of the queries move. Detailed descriptions of the two tables are given below:

Moving Object Table (MOT): An *MOT* entry is a tuple $(i_o, i_q, \overline{p}, \overline{v}, t, B_{msb}, P_{cm}, V_{ch})$ and stores information regarding a moving object. Here, i_o is the moving object identifier, i_q is the query identifier of the moving query whose focal object's identifier is i_o, i_q is *null* if no such moving query exists, $\overline{p}$ is the last received position, $\overline{v}$ is the last received velocity vector of the moving object, t is the time stamp of the motion updates ($\overline{p}$ and $\overline{v}$) received from the moving object, B_{msb} is the *MSB* of the moving object, P_{cm} is an estimate on the period of constant motion of the object, and V_{ch} is a Boolean variable indicating whether the object has changed its motion function since the last query evaluation phase.

Moving Query Table (MQT): An *MQT* entry is a tuple $(i_q, \overline{p}, \overline{v}, r, t, B_{msb}, P_{cm}, V_{ch})$ and stores information regarding a moving query. Here, i_q is the moving query identifier; $\overline{p}$ and $\overline{v}$ are the last received position and the last received velocity vector of the query's focal object, respectively; t is the time stamp of the motion updates ($\overline{p}$ and $\overline{v}$) received from the focal object, r is the radius of the moving query's spatial region, B_{msb} is the *MSB* of the moving query, P_{cm} is an estimate of the period of constant motion of the object, and V_{ch} is a Boolean variable indicating whether or not the focal object has changed its motion function since the last query evaluation phase. Note that the information in MQT is to some extent redundant with respect to *MOT*. However, the redundant information is required during the moving query table scan. Without redundancy, we would need to look them up from the moving object table, which could be costly.

The *MOT* and *MQT* table entries are updated whenever new motion updates are received from the moving objects. The P_{cm} entries are updated using a simple weighted running average. The details are given in Algorithm 1 (Fig. 7). Assuming that moving objects decide whether or not they should send new motion updates at every P_{mu} seconds (called the *motion update period*), one of our aims is to perform a single query evaluation phase in less than P_{mu} seconds in order not to miss any motion updates, i.e., having $P_s \leq P_{mu}$. If, under the available resources, a given implementation of MAI is unable to perform the query evaluation with $P_s \leq P_{mu}$ satisfied, then the query evaluation period P_s has to be increased, i.e., query evaluation has to be performed less frequently. Since the effects of motion updates are reflected in the query results during the next query evaluation step, false positives and false negatives arise between query reevaluations more frequently for larger P_s values. However, this problem is not specific to MAI. In general, when the available resources are not sufficient to handle all queries and position updates in real time, false positives and negatives will temporarily arise in the query results. When we have $P_s \leq P_{mu}$, then it is at least guaranteed that no motion updates are missed.

Although the moving object and query tables increase the storage requirements of the proposed solution, for most cases the server already contains tables corresponding to all objects and all queries. The object table may contain detailed information about various object attributes and the query table may contain attributes of the queries. In the worst case, where all of the objects and all of the queries are moving, we can expect the size of the database to double due to the inclusion of *MOT* and *MQT*. However, we feel that such an increase is acceptable when the improvement in performance is considered.

Fig. 8 gives an overall sketch of the query evaluation process. At each query evaluation phase, we need to perform *query table scan* and *object table scan*. The scan algorithms presented in the next section describe how these two tasks are performed.

4.4.2 The Scan Algorithms

At each query evaluation phase, two scans are performed. The first scan is on the moving object table, *MOT*, and the second scan is on the moving query table, *MQT*. The aim of the *MOT* scan is to update the $Index_o^{msb}$ and to incrementally update some of the query results by performing searches on the $Index_q^{msb}$. The aim of the MQT scan is to update the $Index_q^{msb}$ and to recalculate some of the query results by performing searches on the $Index_o^{msb}$.

Algorithm 2: Moving Object Table Scan

1: **for all** $e = \langle i_o, i_q, \overline{p}, \overline{v}, t, B_{msb}, P_{cm}, V_{ch} \rangle \in MOT$ **do**
2: $t_c \leftarrow$ current time
3: {Calculate the new object position}
4: $e.\overline{p} \leftarrow e.\overline{p} + (t_c - e.t) * e.\overline{v}$
5: $e.t \leftarrow t_c$
6: {B_{inv} is true iff there is MSB invalidation}
7: $B_{inv} \leftarrow e.\overline{p} \notin e.B_{msb}$
8: {If no MSB invalidation and no velocity vector change}
9: **if** $\neg B_{inv} \wedge \neg e.V_{ch}$ **then**
10: **continue** {Nothing to be done}
11: **end if**
12: $B_{old} \leftarrow e.B_{msb}$
13: **if** $e.V_{ch}$ **then**
14: $e.V_{ch} \leftarrow$ **false**
15: **if** $\neg B_{inv}$ **then**
16: {Update the data associated with $e.i_o$ in $Index_o^{msb}$}
17: $Index_o^{msb}.updateData(i_o, \langle e.\overline{p}, e.\overline{v}, e.t \rangle)$
18: **end if**
19: **end if**
20: **if** B_{inv} **then**
21: $\alpha \leftarrow \alpha\beta Table.lookup(|e.\overline{v}|, e.P_{cm})$
22: $e.B_{msb} \leftarrow Rect(e.\overline{p}, e.\overline{p} + \alpha * e.\overline{v})$
23: {Update the entry associated with $e.i_o$ in the $Index_o^{msb}$}
24: $Index_o^{msb}.update(i_o, e.B_{msb}, \langle e.\overline{p}, e.\overline{v}, e.t \rangle)$
25: **end if**
26: {Search $Index_q^{msb}$ using the old MSB}
27: $Q_o \leftarrow Index_q^{msb}.search(B_{old})$
28: {Search (with predictive results) $Index_q^{msb}$ using the new MSB}
29: $Q_n \leftarrow Index_q^{msb}.search(e.B_{msb}, e.\overline{p}, e.\overline{v}, e.t)$
30: **for all** $s = \langle i_q, t_i = [t_{is}, t_{ie}] \rangle \in Q_n$ **do**
31: add $\langle e.i_o, t_i \rangle$ into $PQR(s.i_q)$
32: remove $s.i_q$ from Q_o
33: **end for**
34: **for all** $i_q \in Q_o$ **do**
35: remove $e.i_o$ from $PQR(s.i_q)$
36: **end for**
37: **end for**

Algorithm 3: Moving Query Table Scan

1: **for all** $e = \langle i_q, \overline{p}, \overline{v}, r, t, B_{msb}, P_{cm}, V_{ch} \rangle \in MQT$ **do**
2: $t_c \leftarrow$ current time
3: $e.\overline{p} \leftarrow e.\overline{p} + (t_c - e.t) * e.\overline{v}$
4: $e.t \leftarrow t_c$
5: {Calculate the new query MBR}
6: $B_{qp} \leftarrow Rect(e.\overline{p} - (e.r, e.r), e.\overline{p} + (e.r, e.r))$
7: {B_{inv} is true iff there is MSB invalidation}
8: $B_{inv} \leftarrow B_{qp} \notin e.B_{msb}$
9: {If no MSB invalidation and no velocity vector change}
10: **if** $\neg B_{inv} \wedge \neg e.V_{ch}$ **then**
11: **continue** {Nothing to be done}
12: **end if**
13: **if** $e.V_{ch}$ **then**
14: $e.V_{ch} \leftarrow$ **false**
15: **if** $\neg B_{inv}$ **then**
16: {Update the data associated with $e.i_q$ in the $Index_q^{msb}$}
17: $Index_q^{msb}.updateData(i_q, \langle e.\overline{p}, e.\overline{v}, e.r, e.t \rangle)$
18: **end if**
19: **end if**
20: **if** B_{inv} **then**
21: $\beta \leftarrow \alpha\beta Table.lookup(|e.\overline{v}|, e.P_{cm})$
22: $\overline{p_f} \leftarrow e.\overline{p} + \beta * e.\overline{v}$
23: $e.B_{msb} \leftarrow Rect(e.\overline{p} - sign(e.\overline{v}) * e.r, \overline{p_f} + sign(e.\overline{v}) * e.r)$
24: {Update the entry associated with $e.i_q$ in the $Index_q^{msb}$}
25: $Index_q^{msb}.update(i_q, e.B_{msb}, \langle e.\overline{p}, e.\overline{v}, e.r, e.t \rangle)$
26: **end if**
27: $PQR(e.i_q) \leftarrow \emptyset$
28: {Search (with predictive results) $Index_o^{msb}$ using the query MSB}
29: $O_n \leftarrow Index_o^{msb}.query(e.B_{msb}, e.\overline{p}, e.\overline{v}, e.r, e.t)$
30: **for all** $s = \langle i_o, t_i = [t_{is}, t_{ie}] \rangle \in O_n$ **do**
31: add $\langle s.i_o, t_i \rangle$ into $PQR(e.i_q)$
32: **end for**
33: **end for**

Fig. 9. Object table and query table scans.

MOT Scan. During the MOT scan, when processing an entry, we first check whether the associated object of the entry has invalidated its *MSB* (using $\overline{p}, \overline{v}, t,$ and B_{msb}) or changed its motion function since the last query evaluation period (based on V_{ch}). (See Fig. 9a.) If none of these has happened, we proceed to the next entry *without performing any operation* on the spatial *MSB* indexes. Otherwise, we first update the $Index_o^{msb}$. In case there is an *MSB* invalidation, a new *MSB* is calculated for the object and the $Index_o^{msb}$ is updated. The α value used for calculating the new *MSB* is selected adaptively, using $|\overline{v}|$ and P_{cm} (see Section 4.5.2 for further details). If there has been a motion function change, the data associated with the entry of the object's *MSB* in the $Index_o^{msb}$ is also updated. Once the $Index_o^{msb}$ is updated, two searches are performed on the $Index_q^{msb}$. First, using the old *MSB* of the object, the $Index_q^{msb}$ is searched and all the queries whose *MSB*s intersect with the old *MSB* of the object are retrieved. The object is then removed from the results of those queries (if it is already in). Then, a second search is performed with the newly calculated *MSB* of the object, and all queries whose *MSB*s intersect with the new *MSB* of the object are retrieved. For all those queries, result prediction is performed against the object. Last, the query result entries obtained from the prediction with nonempty time intervals are added into their associated query results.

MQT Scan. During the MQT scan, when processing a query entry we first check whether the associated query of the entry has invalidated its *MSB* (using $\overline{p}, \overline{v}, r, t,$ and B_{msb}) or its focal object has changed its motion function since the last query evaluation phase (based on V_{ch}). (See Fig. 9b) If none of these has happened, we proceed to the next entry *without performing any operation on the spatial indexes*. Otherwise, we first update the $Index_q^{msb}$. In case there is an MSB invalidation, a new MSB is calculated for the query and the $Index_q^{msb}$ is updated. The β value used for calculating the new *MSB* is selected adaptively, using $|\overline{v}|$ and P_{cm} (see Section 4.5.2 for details). If there has been a motion function change, the data associated with the entry of the query's *MSB* in the $Index_q^{msb}$ is also updated. Once the $Index_q^{msb}$ is updated, a single search is performed on the $Index_o^{msb}$ with the newly calculated *MSB* of the query. All objects whose *MSB*s intersect with the new query *MSB* are retrieved. For all those objects, result prediction is performed against the query. The predictive query result entries with nonempty time intervals are added into the query result and all old query results are removed.

Note that after the *MOT* scan, all results are correct for the queries whose *MSB*s are not invalidated and whose focal objects have not changed their motion functions. For queries that have invalidated their *MSB*s or whose focal objects have changed their motion functions, the query results are recalculated during the *MQT* scan. Therefore, all of the query results are up-to-date after the *MQT* scan, given that *MOT* scan is performed first. The order of the scans can be reversed with some minor modifications.

Between query reevaluations, false positives and negatives may arise in the query results. False positives may only arise for objects and queries whose motion functions

TABLE 2
Symbols and Their Meanings

P_s	scan period	R_{mq}	average moving query radius
P_{cm}	avg. period of constant motion	L_{sq}	average static query side length
N_o	number of objects	V_a	average moving object speed
N_{mo}	number of moving objects	A	area of the region of interest
N_q	number of queries	α	MSB parameter for objects
N_{mq}	number of moving queries	β	MSB parameter for queries

have changed since the last query evaluation step. This is because when no motion updates take place, *PQR*s are accurate and can predict the departure of objects from the query regions correctly. On the other hand, false negatives may take place when some of the objects enter into *MSB*s of some queries between query reevaluations. This happens more frequently when P_s is large. Since we encourage performing query reevaluations as frequently as possible, large P_s values are unlikely.

4.5 Setting α and β Values

The α and β parameters used for calculating *MSB*s can be set based on the motion behavior of the objects, in order to achieve more efficient query evaluation. There are two important characteristics of object motions: 1) *the speed of the object* and 2) *the period of constant motion of the object* (i.e., the length of the time period it takes for the motion function to change). For instance, for a query whose focal object changes its motion function frequently, it may not be a good idea to perform too much prediction. Thus, the β value for this query's *MSB* should be kept smaller. However, for an object with high speed, a small α value may not be appropriate, as it may cause frequent *MSB* invalidations. As a result, it is important to design a motion-adaptive method that can set the values of α and β parameters adaptively. A common approach to runtime parameter setting is to develop an analytical model and use it to guide the runtime selection of the best parameter settings. We develop an analytical model for estimating the IO cost of performing query evaluation. This model is used as the guide to build an offline computed $\alpha\beta Table$, giving the best α and β values for different value pairs of speed and period of constant motion of a moving object.

4.5.1 Analytical Model for IO Estimation

We develop an analytical model for estimating the IO cost of performing query evaluation, i.e., the two scans performed at each query evaluation phase. The formulations in this section are derived based on the average values for the speed and the period of constant motion of a moving object. For the purpose of offline $\alpha\beta Table$ creation, the associated speed and period of constant motion values are taken from the table cells. Table 2 lists some of the symbols used in this section and their meanings.

Let A_{mo} denote the average area of a moving object *MSB* and A_{mq} denote the average area of a moving query *MSB*. Denoting the average object speed as V_a, based on the definition of *MSB*s we have:

$$A_{mo} = (\alpha * V_a/\sqrt{\pi})^2, \text{and}$$
$$A_{mq} = (\beta * V_a/\sqrt{\pi} + 2 * R_{mq})^2.$$

The derivation of A_{mo} follows from the fact that the side of a moving object *MSB* has an average size of α times the average speed of the object on the side's direction. Averaging over all possible angles for the velocity vector, we have

$$A_{mo} = (\alpha * V_a)^2 * \frac{1}{2*\pi}\int_0^{2*\pi} |\sin x| * |\cos x|\, dx$$
$$= (\alpha * V_a/\sqrt{\pi})^2.$$

The derivation for the moving query *MSB*s follow a similar formulation, with the exception that the diameter of the query, denoted by $2 * R_{mq}$, is also included in the equation.

Let A_o denote the average size of the object bounding boxes stored in the $Index_o^{msb}$ (static objects are assumed to have a box with zero area) and A_q denote the average size of the query bounding boxes stored in the $Index_q^{msb}$. Then, we have

$$A_o = A_{mo} * \frac{N_{mo}}{N_o}, \text{ and}$$
$$A_q = \frac{N_{mq}}{N_q} * A_{mq} + \left(1 - \frac{N_{mq}}{N_q}\right) * L_{sq}^2.$$

The derivation of A_o follows from the fact that N_{mo}/N_o fraction of the objects (that are moving objects) have an average *MSB* size of A_{mo} and the rest (stationary objects) have an *MSB* size of 0. The derivation of A_q follows similarly. Stationary queries, which form N_{mq}/N_q fraction of all queries, have an average *MSB* size of L_{sq}^2, where L_{sq} is the average side length of a static range query. On the other hand, moving queries, that form N_{mq}/N_q fraction of all queries, have an average *MSB* size of A_{mq}.

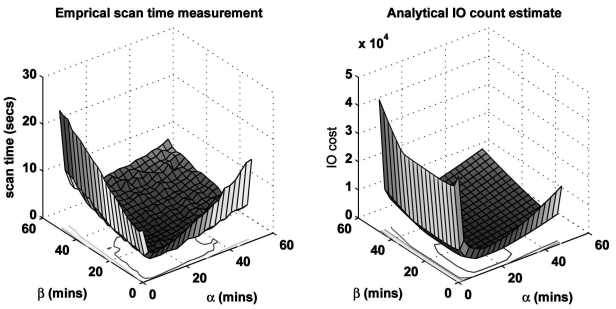
Given this information, the following four quantities can be analytically derived based on well-studied *R-tree* cost models [26]: node IO cost during the processing of

1. an object-table entry for updating the $Index_o^{msb}$, C_o^u;
2. an object-table entry for searching the $Index_q^{msb}$, C_o^s;
3. a query-table entry for updating the $Index_q^{msb}$, C_q^u; and
4. a query-table entry for searching the $Index_o^{msb}$, C_q^s.

Let N_o^{vc} denote the expected value of the number of distinct objects causing velocity change events during one scan period and let N_q^{vc} denote the expected value of the number of distinct queries causing velocity change events during one scan period. If $P_s/P_{cm} < 1$, only some of the moving objects will cause velocity change events. Hence, we have

$$N_o^{vc} = N_{mo} * \min\left(1, \frac{P_s}{P_{cm}}\right), \text{ and}$$
$$N_q^{vc} = N_{mq} * \frac{N_o^{vc}}{N_{mo}}.$$

The derivation of N_q^{vc} follows from the fact that a moving query causes a velocity change event only if its focal object causes a velocity change event and that only N_o^{vc}/N_{mo} fraction of the moving objects cause velocity change events.



(a) (b)

Fig. 10. (a) Experimental query evaluation time and (b) analytical node IO estimate.

Let N_o^{bi} denote the expected value of the number of objects causing box invalidations during one scan period and N_q^{bi} denote the expected value of the number of queries causing box invalidations during one scan period. If $P_s/\alpha < 1$, only some of the moving objects will cause box invalidations. Similarly, if $P_s/\beta < 1$, only some of the moving queries will cause box invalidations. Then, we have

$$N_o^{bi} \approx \min\left(1, \frac{P_s}{\alpha}\right) * N_{mo}, \text{ and}$$
$$N_q^{bi} \approx \min\left(1, \frac{P_s}{\beta}\right) * N_{mq}.$$

Let N_{mot} denote the expected value of the number of entries in the object table that caused velocity change or box invalidation events and let N_{mqt} denote the expected value of the number of entries in the query table that caused velocity change or box-invalidation events. Assuming that an object causes a velocity change event independent of whether it has caused an *MSB* invalidation and similarly assuming that a query causes a velocity change event independent of whether it has caused an *MSB* invalidation, we have:

$$N_{mot} = N_o^{vc} + N_o^{bi} - N_o^{bi} * \frac{N_o^{vc}}{N_{mo}}, \text{ and}$$
$$N_{mqt} = N_q^{vc} + N_q^{bi} - N_q^{bi} * \frac{N_q^{bi}}{N_{mq}}.$$

Finally, the total IO cost for the periodic scan, C_{io}, can then be calculated, considering that for an entry of *MOT* that requires processing due to velocity change or *MSB* invalidation, an update on the $Index_o^{msb}$ and two searches on the $Index_q^{msb}$ are needed, and for an entry of MQT that requires processing due to velocity change or MSB invalidation, an update on the $Index_q^{msb}$ and a search on the $Index_o^{msb}$ are needed:

$$C_{io} = N_{mot} * (C_o^u + 2 * C_o^s) + N_{mqt} * (C_q^u + C_q^s). \quad (1)$$

4.5.2 $\alpha\beta Table$ and Adaptive Parameter Selection

The cost function developed in this section has a global minimum that optimizes the IO cost of the query evaluation. We build an offline computed $\alpha\beta Table$, which gives the optimal α and β values for different value pairs of object speed ($\bar{v}$) and period of constant motion (P_{cm}), calculated using the cost function we have developed. We implement the $\alpha\beta Table$ as a 2D matrix, whose rows correspond to different object speeds and whose columns correspond to different periods of constant motion, and the entries are optimal (α, β) pairs. Recall that, as discussed in Section 4.4, when we calculate the MSBs of moving objects and moving queries, we already have the estimates on periods of constant motion and speeds of all moving objects including the focal objects of the moving queries. We can decide the best α and β values to use during MSB calculation by performing a single lookup from the offline computed $\alpha\beta Table$.

TABLE 3
A Sampled Subset of the $\alpha\beta Table$ from the Experiment of Fig. 14 in Section 6

period of constant motion \ velocity	10	20	30	40	50	60	70	80	90
0.01	(.98,.75)	(.98,.13)	(.78,.08)	(.63,.07)	(.53,.05)	(.47,.05)	(.42,.05)	(.37,.03)	(.33,.03)
0.05	(.98,.75)	(.98,.18)	(.98,.13)	(.90,.12)	(.73,.10)	(.62,.10)	(.53,.08)	(.47,.08)	(.42,.08)
0.10	(.98,.75)	(.98,.23)	(.98,.18)	(.97,.17)	(.75,.15)	(.62,.13)	(.52,.13)	(.43,.12)	(.38,.12)
0.15	(.98,.75)	(.98,.28)	(.98,.23)	(.93,.22)	(.72,.18)	(.58,.17)	(.50,.17)	(.43,.17)	(.40,.17)
0.20	(.98,.75)	(.98,.33)	(.98,.28)	(.88,.25)	(.70,.22)	(.58,.22)	(.52,.22)	(.47,.22)	(.43,.22)
0.25	(.98,.75)	(.98,.38)	(.98,.32)	(.87,.28)	(.72,.27)	(.62,.27)	(.55,.27)	(.47,.27)	(.40,.27)
0.30	(.98,.75)	(.98,.42)	(.98,.35)	(.88,.32)	(.73,.32)	(.65,.32)	(.53,.32)	(.43,.32)	(.43,.22)
0.35	(.98,.75)	(.98,.45)	(.98,.37)	(.92,.37)	(.77,.37)	(.62,.37)	(.50,.37)	(.50,.23)	(.43,.22)
0.40	(.98,.75)	(.98,.48)	(.98,.42)	(.93,.42)	(.77,.42)	(.58,.42)	(.55,.30)	(.50,.23)	(.45,.20)
0.45	(.98,.75)	(.98,.52)	(.98,.47)	(.98,.47)	(.73,.47)	(.65,.33)	(.55,.30)	(.50,.23)	(.47,.18)
0.50	(.98,.78)	(.98,.55)	(.98,.52)	(.97,.52)	(.70,.52)	(.65,.33)	(.55,.30)	(.52,.22)	(.52,.17)

Fig. 10a plots the average time it takes to perform one complete query evaluation phase (labeled as *total query evaluation time*) as a function of α and β. These values are from the actual implementation of motion-adaptive indexing. Fig. 10b plots the analytical node IO count estimate of performing one query evaluation phase as a function of α and β. Two important observations can be obtained by comparing these graphs. First, they show that the IO cost is dominant on the time it takes to perform query evaluation, as the node IO count graph highly determines the shape of the query evaluation time graph. Second, the optimal values of α and β calculated using the analytical cost function indeed result in faster query evaluation.

In Table 3, we give a sampled subset of the $\alpha\beta Table$ that is used in the experiment reported in Fig. 12 of Section 6. The actual table covers a larger range and has a higher resolution. Each entry in the table is in the form (α, β). We make two observations from Table 3. First, with increasing object speeds, the optimal α and β values decrease. This is because for high speeds the α and β parameters should be kept small in order to avoid large MSBs, which will cause high overlap and increase the cost of spatial index operations. Second, with decreasing period of constant motion, the optimal α and β values decrease. This is because large MSBs are undesirable when the predictability is poor (period of constant motion is small), since they will result in a larger number of invalidated PQRs and thus increased IO cost. We will provide performance results on the improvement provided by the adaptive parameter selection in Section 6.3.

5 Evaluating Moving kNN Queries with Motion Adaptive Indexing

Moving continual k-nearest neighbor (kNN) queries over moving objects can be evaluated using the main mechanisms employed for moving range query evaluation. A moving kNN query is defined similar to a moving range query, except that instead of a range, the parameter k is specified for retrieving the k nearest neighbors of the focal object of the query.

A unique feature of our motion-adaptive indexing scheme is its ability to efficiently process both continual moving *range* queries and continual moving kNN queries. Note that, for a mobile database system that has to manage both range MCQs and kNN MCQs, solutions that are exclusive to kNN queries will introduce extra overhead, since the indexes and data structures are not shared with the range query evaluation component, further exacerbating the problem of high index maintenance cost in moving object databases. In contrast, our solution uses a common framework to support both range and kNN queries, so that workloads that are mixtures of kNN queries and range queries are efficiently handled. In order to extend the motion-adaptive indexing developed for evaluating moving range queries to the evaluation of moving kNN queries, we introduce the concept of *safe radius* and two mechanisms: *guaranteed safe radius* and *optimistic safe radius*. To evaluate kNN queries with the use of safe radii, we need to make the following three changes:

1. During the MQT table scan, when a query invalidates its MSB or changes its motion function, we calculate a *safe radius*, which is guaranteed to contain at least k moving objects until the next time the safe radius is calculated (β is an upper bound for this time). Then, the kNN query is installed as a standard MCQ with its range equal to the safe radius.
2. Instead of storing time intervals in query result entries, we store the distance of the objects from the focal object of the query as a function of time.
3. At the end of each query evaluation phase, results are sorted based on their distances to their associated focal objects by using the distance functions stored within the query result entries. The top k result entries are then marked as the current results.

The important step here is to calculate a safe radius that will make sure that at least k objects will be contained within the safe radius during the next t time units. We propose two different approaches to tackle this problem: the guaranteed safe radius (GSR) and the optimistic safe radius (OSR).

The guaranteed safe radius approach retrieves the current k nearest neighbors, and for each object in the list calculates the maximum possible value the distance between the object and the focal object of the query can take *at the end* of the next t time units. This can be calculated using the focal object's motion function and the *upper bounds on the maximum speeds* of these k nearest neighbor objects. The maximum of these k calculated distances will give the safe radius. However, there are two problems. First, it requires us to know the upper bounds on the speeds of moving objects. Second, the calculated safe radius may become unnecessarily large, negatively affecting the performance.

The optimistic safe radius approach retrieves the current k nearest neighbors, and for each object in the list calculates the maximum value of the distance between the object and the focal object of the query can take *throughout* the next t time units, assuming that the objects will not change their motion functions during this time. For each of the k objects, this calculation can be done using the *current motion function* of the object and the motion function of the query's focal object. The maximum of these k calculated distances will give the safe radius. This approach guarantees that k objects will be contained within the safe radius during the next t time units under the assumption that the initial set of k nearest neighbors do not change their motion functions during this period. When using this approach, if the number of objects in the result of a kNN query turns out to be smaller than k, we fall back to the traditional spatial index kNN search plan for that query until the next time a new safe radius is calculated.

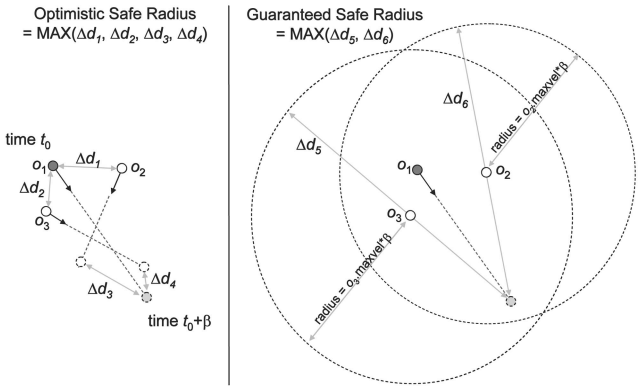


Fig. 11. Illustration of optimistic and guaranteed safe radius calculation for 2NN queries.

Fig. 11 illustrates how safe radii are calculated with an example 2NN query, where the focal object is o_1 and the two nearest neighbors at time t_0 are objects o_2 and o_3. The safe radius is calculated to be valid during the next β time units. We will provide the performance comparison of *guaranteed safe radius (GSR)* and *optimistic safe radius (OSR)* in Section 6.

6 Experimental Results

This section describes five sets of experiments, which are used to evaluate our solution. The first set of experiments compares the performance of motion-adaptive indexing against various existing approaches. The second set of experiments illustrates the advantages of adaptive parameter selection over fixed parameter setting on the sizes of bounding boxes. The third set of experiments studies the effect of skewed data and query distribution on query evaluation performance. The fourth set of experiments analyzes the scalability of the proposed approach with respect to queries with varying sizes of spatial regions, varying percentages of moving queries, and varying numbers of objects. Finally, the fifth set of experiments present the effectiveness of the motion-adaptive approach to evaluating moving continual kNN queries over moving objects.

6.1 System Parameters and Setup

In the experiments presented in the rest of the paper, the parameters take their default values listed in Table 4, when not specified otherwise. Based on the default values, 50 percent of the objects are moving and the remaining 50 percent are static. Similarly, 50 percent of the queries are moving and the remaining 50 percent are static. Different percentages of moving queries are studied in Section 6.6. Moving queries are assigned range values from the list $\{5, 4, 3, 2, 1\}$ (in miles) using a Zipf distribution with parameter 0.6. Static queries are assigned side range values from the list $\{8, 7, 5, 4, 2\}$ (in miles) using a Zipf distribution with parameter 0.6.

The default object density is taken in accordance with previous work [19], [20]. Objects and queries are randomly distributed in the area of interest except in Section 6.5, where we consider skewed distributions. Objects that belong to

TABLE 4
System Parameters

Parameter	Default value / Range
area of the region of interest	500000 sq. miles
number of objects	50000 / [50K,200K]
percentage of moving objects	50
number of queries	5000 / [2.5K,20K]
percentage of moving queries	50 / [0,100]
moving query range distribution	$\{5, 4, 3, 2, 1\}$ miles with Zipf param 0.6
static query side range distribution	$\{8, 7, 5, 4, 2\}$ miles with Zipf param 0.6
period of constant motion	mean 5 minutes, geometrically distributed
moving object speed	between 0-150 miles/hour uniformly random
scan period	30 seconds
motion update period	30 seconds

different classes with strictly varying movement behaviors are considered in Section 6.3. The paths followed by the objects are random, i.e., each time a motion function update occurs, a random direction and a random speed are chosen. The object speeds are selected from the range $(0, 150]$ (in miles/hour) uniformly at random. Table 4 gives details of other important system parameters. We vary the values of many system parameters to study their effects on the performance.

For R*-trees, a 101-node LRU buffer is used with 4-KByte page size. Branching factor of the internal tree nodes is 100 and the fill factor is 0.5. Relative merits of our techniques shown in the rest of the section are also valid under scenarios with large buffer sizes (which effectively makes it a main memory algorithm); however, we do not report those results. All experiments are performed using R*-trees, except that in Section 6.5 a static grid-based spatial index implementation is used for comparison purposes.

We compare the performance of motion-adaptive indexing against various existing approaches, in terms of query evaluation time and node IO counts. The approaches used for comparison are *brute force (BF)*, *object-only indexing (OI)*, *query-only indexing (QI)*, *object-and-query indexing (OQI)*, *motion-adaptive indexing (MAI)*, and *object indexing with MSBs (OIB)*. The brute force calculation is performed by scanning through the objects. During the scan, all queries are considered against each object in order to calculate the results. The *OI* approach uses an object index which is updated for all objects that have moved since the last query evaluation phase[1] and searched for all queries in order to evaluate the query results. The *QI* approach uses a query index, which is updated for all queries that have moved since the last query evaluation step and is searched for all object positions in order to update the query results incrementally. *OQI* is a stripped-down version of *MAI* without *MSB*s and *PQR*s. *OIB*s is similar to pure object-only indexing, except that the motion sensitive boxes are used instead of object positions in the spatial index (without the *PQR*s).

6.2 Performance Comparison

Fig. 12 plots the total query evaluation time for a fixed number of objects (50 K) with a varying number of queries (2.5 K to 20 K). The horizontal line in the figure represents the scan period. We consider a query evaluation scheme as acceptable when the total query evaluation time is less than the scan period. Note that the scan period, P_s, is set to be equal to the motion update period, P_{mu}, in this set of experiments. Fig. 13 plots the query evaluation node IO count for the same setup. The node IO is divided into four different components:

1. node IO due to object index update,
2. node IO due to object index search,
3. node IO due to query index update, and
4. node IO due to query index search.

Each component is depicted with a different shade in Fig. 13. Several observations can be obtained from Fig. 12 and Fig. 13.

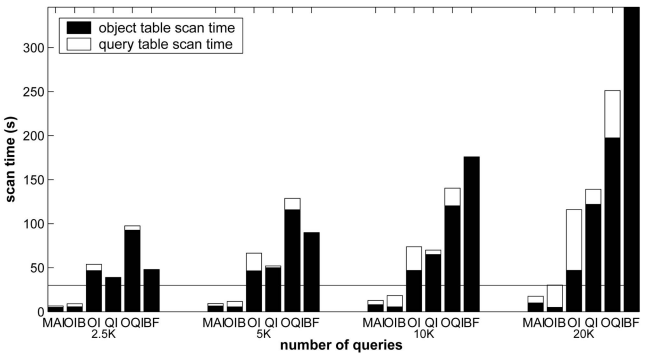


Fig. 12. Query evaluation time.

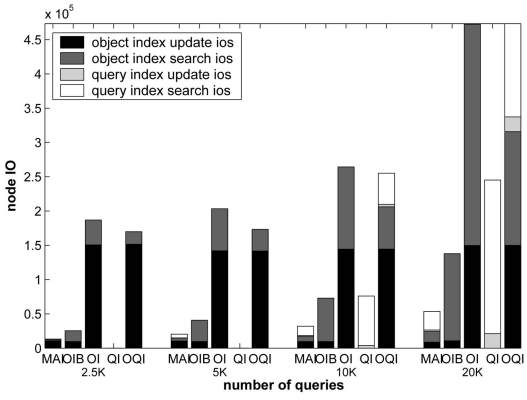


Fig. 13. Query evaluation node IO.

1. Although update-efficient object indexes exist [20], [24], we show that their use does not change our conclusion for large or moderate numbers of queries, in which cases search cost is the dominant factor.

First, the approaches with an object index that is updated for all moving objects do not perform well when the number of queries is small. This is clear from the poor performances of *OI* and *OQI* for 2.5 K queries, as shown in Fig. 12. The reason is straightforward. The cost of updating the object index dominates when the number of queries is small. This can also be observed by the obejct index update component of the *OI* in Fig. 13. However, there are also significant costs for searching the object index for the *OI* approach. These costs dominate the total IO cost when the number of queries is large (see the case of 20 K queries in Fig. 13). This points out an important fact: *Although it is possible to reduce the cost of updating the object index (for instance, by using a TPR-tree-based object index [20], [24]), MAI still performs significantly better than such an obejct index-based approach.*

Second, the approaches with a query index that is searched for a large number of objects do not perform well for a large number of queries. This is clear from the poor performances of *QI* and *OQI* for 20 K queries, as shown in Fig. 12. This is due to the fact that the cost of searching the query index dominates when the number of queries is large. This can also be observed by the query index search component of the *QI* in Fig. 13. Note that, for a small number of queries, the node IO count for *QI* appears as 0, because the query index fits into the *LRU* buffer.

Third, the brute force approach performs relatively well compared to *OQI* and slightly better than *OI* when the number of queries is small (2.5 K), as shown in Fig. 12. Obviously, *BF* does not scale with the increasing number of queries, since the computational complexity of the brute force approach is $O(N_o * N_q)$, where N_o is the total number of objects and N_q is the total number of queries. Although *OQI* seems to be a consistent loser when compared to other indexing approaches, it is interesting to note that the motion-adaptive indexing is built on top of it and performs better than all other approaches.

Finally, it is worth noting that only *MAI* manages to provide good enough performance to satisfy $P_s \leq P_{mu}$ under all conditions. MAI provides around 75-80 percent savings in query evaluation time under all cases when compared to the best competing approach except *OIB*. However, *OIB* performs reasonably well but fails to scale well with increasing number of queries when compared to the proposed *MAI* approach.

6.3 Effect of Adaptive Parameter Selection

In order to illustrate the advantage of adaptive parameter selection, we compare motion-adaptive indexing against itself with static parameter selection. For the purpose of this experiment, we introduce three different classes of moving objects with strictly different movement behaviors. The first class of moving objects change their motion functions frequently (average period of constant motion: 1 minute) and move slowly (maximum speed: 20 miles/hour). The second class of moving objects possess the default properties described in Section 6.1. The third class of moving objects seldom change their motion functions (avgerage period of constant motion: 30 minutes) and move fast (maximum speed: 300 miles/hour). In order to observe the gain from adaptive parameter selection, we set the α and β parameters to the optimal values obtained for moving objects of the second class for the nonadaptive case.

Fig. 14 plots the time and IO cost of query evaluation for *MAI* and a static parameter setting version of *MAI*. The x-axis represents the object class distributions. Hence, 1:1:1 represents the case where the number of objects belonging to different classes are the same. Along the x-axis, we change the number of objects belonging to the second class. 1:0.25:1 represents the case where the number of objects belonging to the first class and the number objects belonging to the third class are both four times the number of objects belonging to the second class. Dually, 1:4:1 represents the case where the second class cardinality is four times those of the other two classes. Total query evaluation times are depicted as lines in the figure and their corresponding values are on the left y-axis. The node IO counts are depicted as an embedded bar chart and their corresponding values are on the right y-axis. There are two important observations from Fig. 14.

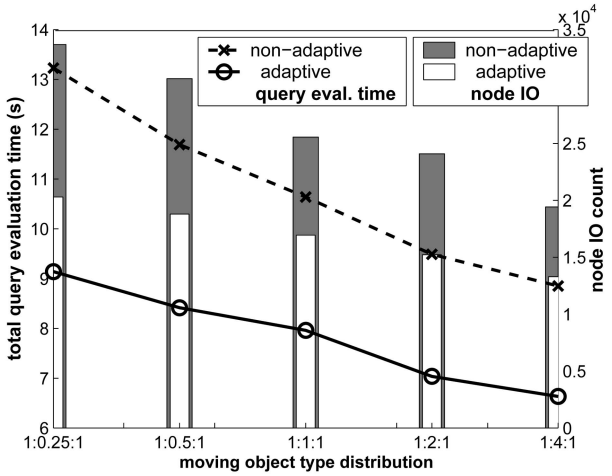


Fig. 14. Performance gain due to adaptive parameter selection.

First, we notice that the adaptive parameter selection has a clear performance advantage. This is clearly observed from Fig. 14, which shows significant improvement provided by motion-adaptive indexing over static parameter setting in both query evaluation time and node IO count.

Second, it is important to note that the objects belonging to the first class or the third class cannot be ignored even if their numbers are small. Even for 1:4:1 distribution, where the second class of objects is dominant, we see a significant improvement with *MAI*. Note that objects belonging to the first and the third class are expensive to handle. The first class of objects are expensive as they cause frequent motion updates which in turn causes more processing during *MOT* and *MQT* scans. The third class of objects are also expensive, as they cause frequent *MSB* invalidation which instigates more processing during *MOT* and *MQT* scans. The fact that both query evaluation time and node IO count are declining along the x-axis shows that it is obviously more expensive to handle the first and the third class of objects.

6.4 Storage Cost

Since *MAI* uses both an object index and a query index, its storage requirements are expected to be larger than the storage requirements of the other alternatives considered in this section. However, given that the processing resources are the limiting factor for handling continuous queries in the mobile object monitoring context, this increase in the storage cost is acceptable considering the savings in IO cost and query evaluation time provided by *MAI*. In Fig. 15, we report the storage cost of the *MAI* approach, relative to other alternatives, for three different settings for the

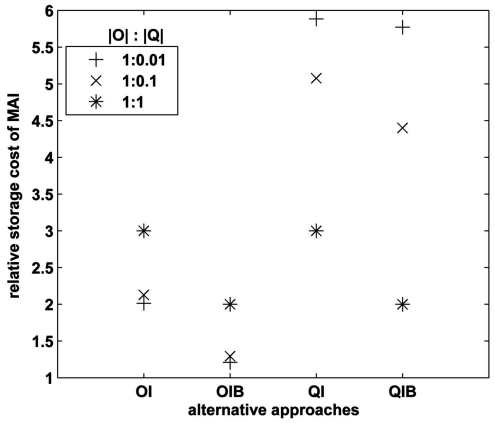


Fig. 15. Storage cost of *MAI* relative to other alternatives.

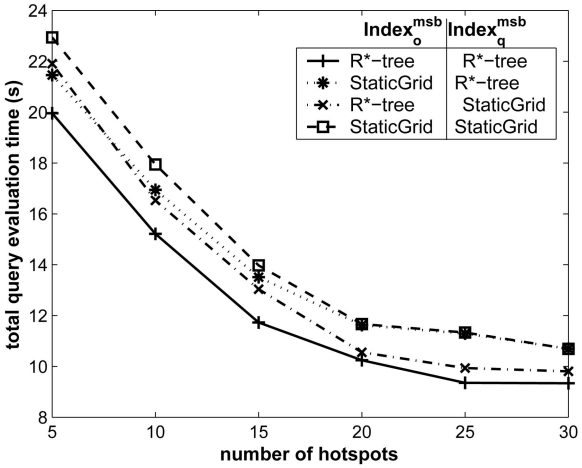


Fig. 16. Effect of data and query skewness on performance.

$|O| : |Q|$ ratio, that are 1:0.01, 1:0.1, and 1:1. We observe from the figure that, relative to *OI* and *OIB*, *MAI* has a storage cost of around three times and 1.25 times for the case of $|O| : |Q| = 0 : 0.01$ and around 2.15 times and 1.35 times for the case of $|O| : |Q| = 0 : 0.1$, respectively. For the extreme case of $|O| : |Q| = 1 : 1$, where the number of queries is equal to the number of objects, we see that *MAI* has a storage cost of around three times and two times relative to *OI* and *OIB*, respectively. In general, the number of queries is expected to be smaller than the number of objects. Thus, it is fair to say that *MAI* has a storage cost that is around two times that of a simple obejct index-based approach. The figure also shows results relative to the *QI* and *QIB* approaches. It is observed that *MAI* incurs up to five times more storage cost compared to *QI*, the worst-case scenario happening when the number of queries is the smallest, that is $|O| : |Q| = 0 : 0.01$. However, given the poor performance of *QI* compared to both *OI* and *MAI*, the savings it provides in terms of storage cost are not of much value.

6.5 Effect of Data and Query Skewness

Our experiments up to now have assumed uniform object-and-query distribution. In this section, we conduct experiments with skewed data and query distributions. We model skewness using two parameters, *number of hot spots (N_h)* and *scatter deviation (d)*. We randomly pick N_h different positions within the area of interest, which correspond to hot-spot regions. When assigning an initial position to an object, we first pick a random hot-spot position from the N_h different hot spots and then place the object around the hot-spot position using a normally distributed distance function on both x and y dimensions with zero mean and d standard deviation. Scatter deviation d is set to 25 miles in all experiments and the number of hot spots is varied to experiment with different skewness conditions. Queries also follows the same distribution with objects. Fig. 17 shows the object and query distribution for $N_h = 5$ and $N_h = 30$.

We also experiment with different spatial indexing mechanisms. We have implemented a static grid-based spatial index, backed up by a B^+-tree with z-ordering [6]. The optimal cell size of the grid is determined based on the workload. The motivation for using a static grid is that with frequently updated data it may be more profitable to use a statically partitioned spatial index that can be easily updated. Actually, previous work done for static range queries over moving objects [11] has shown that using a static grid outperforms most other well-known spatial index structures for in-memory databases. With this experiment, we also investigate whether a similar situation exists in secondary storage based indexing in the context of MCQs.

Fig. 16 plots the total query evaluation time as a function of number of hot spots for different spatial index structures used for $Index_o^{msb}$ and $Index_q^{msb}$. Note that the smaller the number of hot spots, the more skewed the distribution is. Fig. 16 shows that decreasing the number of hot spots quadratically increases the query evaluation time. But even for $N_h = 5$, the query evaluation time does not exceed the query evaluation period. Fig. 16 also shows that R^*-tree performs the best under all conditions.

6.6 Scalability Study

In this section, we study the scalability of the proposed solution with respect to the varying size of query ranges, the varying percentage of moving queries over the total number of spatial queries, and the varying total number of objects. We first measure the impact of the query range and the moving query percentage on the query evaluation performance. We use the *range factor (r_f)* to experiment with different workloads in terms of different query ranges. The query radius and query side length parameters given in Section 6.1 are multiplied by the range factor r_f in order to alter the size of query regions. Note that multiplying the range factor by two in fact increases the area of the query range by four.

Fig. 18 plots the total query evaluation time as a function of moving query percentage for different range factors. As

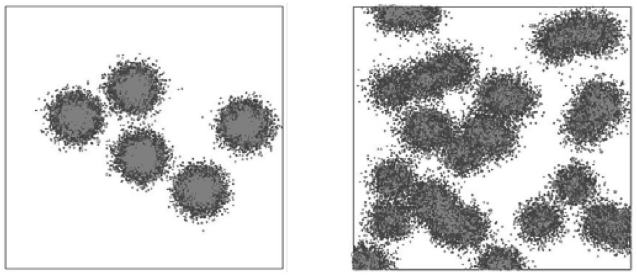


Fig. 17. Query and object distribution for $N_h = 5$ and $N_h = 30$.

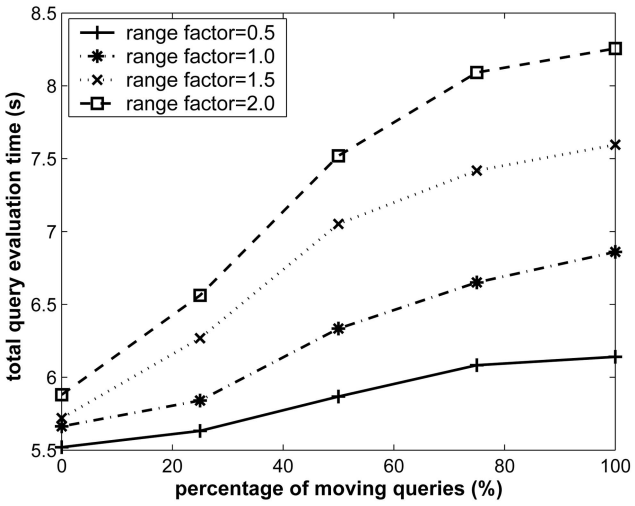


Fig. 18. Effect of query range and moving query percentage on performance.

shown in Fig. 18, the scalability in terms of moving query percentage is extremely good. The slope of the query evaluation time function shows good reduction with increasing percentage of moving objects. Increasing the range factor shows roughly linear increase (with a multiplier that increases with increasing moving query percentage, ≈ 0.25 to ≈ 0.5 for 0 percent to 100 percent) on the query evaluation time.

In Fig. 19, we study the effect of the number of objects on the query evaluation performance. Fig. 19 plots the total query evaluation time as a function of number of objects for different spatial index structures used for $Index_o^{msb}$ and $Index_q^{msb}$. The number of queries is set to its default value of 5 K. From Fig. 19, we observe a linear increase in the query evaluation time with the increasing number of objects. The query evaluation time for 200 K objects is around four times the query evaluation time for 50 K objects for the R^*-tree implementation of $Index_o^{msb}$ and $Index_q^{msb}$, which shows better scalability with increasing number of objects than the static-grid implementation.

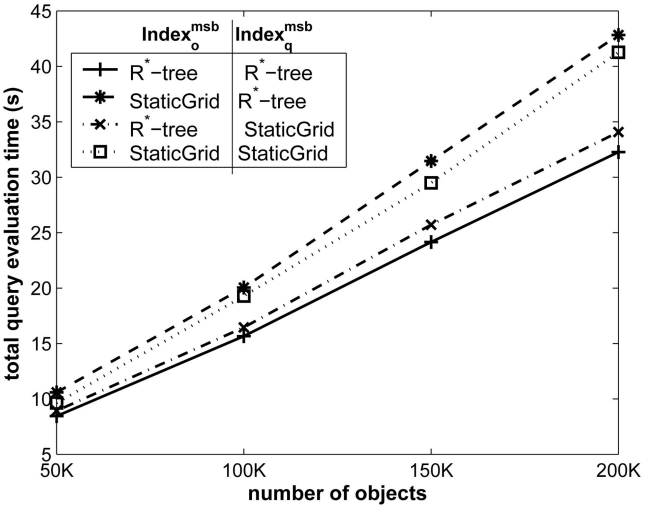


Fig. 19. Effect of number of objects on performance.

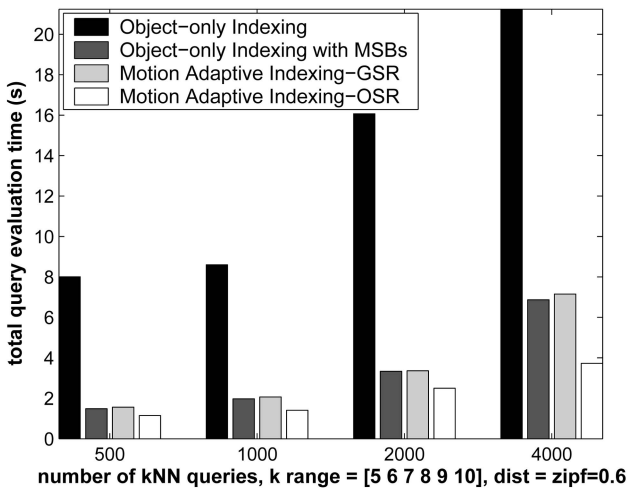


Fig. 20. Total query evaluation time for moving continual kNN queries.

6.7 Performance Comparison for Continual kNN Queries

We compare the performance of MCQ-based moving continual kNN query evaluation against the object-only indexing approach. In object-only indexing, the object index is updated and the kNN queries are evaluated against the updated object index during each query evaluation phase. In this experiment, 10 K objects are used with the same object density (N_o/A) specified in Section 6.1, where 50 percent of the objects are moving with the default motion parameters from Section 6.1. All queries are moving continual kNN queries and the number of queries ranges from 0.5 K to 4 K. The k values of the kNN queries are selected from the list $\{5, 6, 7, 8, 9, 10\}$ using a Zipf distribution with parameter 0.6. Fig. 20 plots the total query evaluation time and Fig. 21 plots the node IO count for different number of objects with different approaches. The node IO count is divided into two components. The lower part shows the node IO due to index searches, where the upper part shows the node IO due to index updates.

Evaluating moving continual kNN queries with motion-adaptive indexing shows significant improvement over an object-only indexing approach. Between the two variations

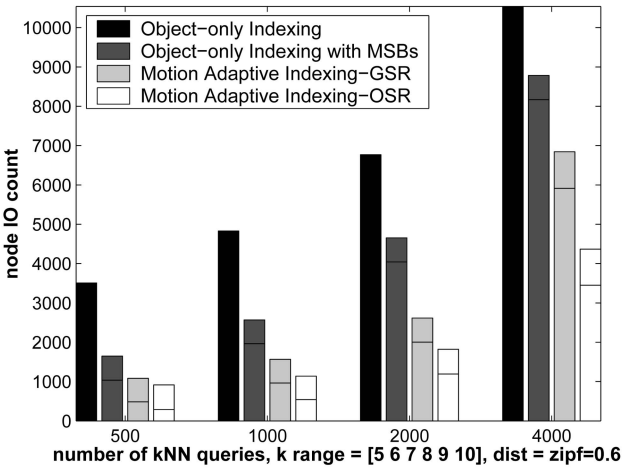


Fig. 21. Node IO count for moving continual kNN queries.

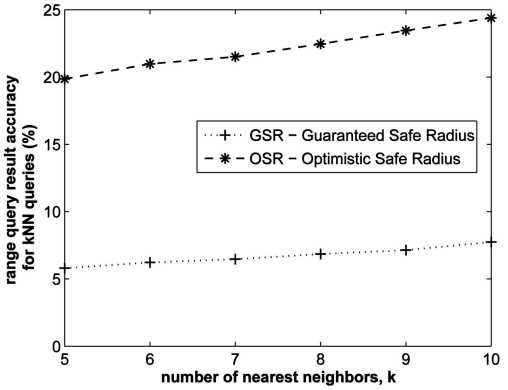


Fig. 22. Range *MCQ* result accuracy for kNN queries.

of safe radius, *OSR* (optimistic safe radius-based approach) performs better than *GSR* (guaranteed safe radius-based approach). Object-only indexing with *MSB*s (*OIB*) slightly outperforms *GSR*. However, *OSR* provides 20 to 40 percent improvement in total query evaluation time over *OIB*.

An interesting statistic is the average result accuracy of the range *MCQ*s used to answer kNN queries, for *GSR* and *OSR* techniques. Concretely, the ratio of the k value specified in the query to the average number of results in the *MCQ* used to answer the query is an important measure to assess the effectiveness of using range queries as a filtering step in answering kNN. In Fig. 22, the range *MCQ* result accuracy for kNN queries is plotted as a function of k for optimistic and guaranteed safe radius techniques. The k values used in the figure are in the range $[5, 10]$. For $k = 5$ and with *OSR*, one-fifth of the results of a range *MCQ* constitute the result of the associated kNN query. This means that the size of the result set of the range MCQ is 25 for a 5NN query, on the average. Importantly, the accuracy of the range *MCQ*s increase with increasing k. For instance, for $k = 10$ and with OSR, one-quarter of the results of a range MCQ constitute the result of the associated kNN query. This increasing trend in accuracy is very useful, since the cost of query evaluation increases with increasing k and it is important that the range MCQs provide good filtering for such costly kNN queries. Fig. 22 also shows that *GSR* performs poorly compared to *OSR*, having a very low accuracy value of 6 percent to 8 percent, where k ranges from 5 to 10.

7 Conclusion

We have presented a system and a motion-adaptive indexing scheme for efficient processing of moving queries over moving objects. Our approach has three unique features. First, we use the concept of *motion-sensitive bounding boxes (MSBs)* to model the dynamic motion behavior of both moving objects and moving queries and we promote indexing less frequently changing *MSB*s together with the motion functions of the objects, instead of indexing frequently changing object positions. This significantly decreases the number of update operations performed on the indexes. Second, we propose using *motion-adaptive* indexing in the sense that the sizes of the *MSB*s can be dynamically adapted to the moving object behavior at the granularity of individual objects. Concretely, we develop a model for estimating the cost of moving query evaluation, and use the analytical model to guide the setting and the adaptation of several system parameters dynamically. As a result, the moving queries can be evaluated faster by performing fewer IOs Finally, we advocate the use of *predictive query results* to reduce the number of search operations to be performed on the spatial indexes. Other important characteristics of our approach include the extension of the motion-adaptive indexing scheme to the evaluation of moving continual kNN queries through the concept of *guaranteed safe radius* and *optimistic safe radius*. We report a series of experimental performance results for different workloads, including scenarios based on skewed object and query distribution, and demonstrate the effectiveness of our motion-adaptive indexing scheme through comparisons with other alternative indexing mechanisms. We have shown that the proposed motion-adaptive indexing scheme is efficient for evaluation of both moving continual *range* queries and moving continual kNN queries.

Acknowledgments

This research is partially supported by grants from US National Science Foundation (NSF) CSR, NSF ITR, US Department of Energy SciDAC, an IBM Faculty Award, an IBM SUR grant, and an HP equipment grant.

References

[1] C.C. Aggarwal and D. Agrawal, "On Nearest Neighbor Indexing of Nonlinear Trajectories," *Proc. ACM Principles of Database Systems (PODS),* pp. 252-259, 2003.

[2] R. Benetis, C.S. Jensen, G. Karciauskas, and S. Saltenis, "Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects," *Proc. Int'l Database Eng. and Applications Symp. (IDEAS),* pp. 44-53, 2002.

[3] Y. Cai and K.A. Hua, "An Adaptive Query Management Technique for Efficient Real-Time Monitoring of Spatial Regions in Mobile Database Systems," *Proc. IEEE Int'l Performance Computing and Comm. Conf. (IPCCC),* pp. 259-266, 2002.

[4] A. Civilis, C.S. Jensen, J. Nenortaite, and S. Pakalnis, "Efficient Tracking of Moving Objects with Precision Guarantees," *Proc. Int'l Conf. Mobile and Ubiquitous Systems (MobiQuitous),* pp. 164-173, 2004.

[5] R.M. Fujimoto, *Parallel and Distributed Simulation Systems.* Wiley-Interscience, 2000.

[6] V. Gaede and O. Gunther, "Multidimensional Access Methods," *ACM Computing Surveys,* vol. 30, no. 2, pp. 170-231, 1998.

[7] B. Gedik and L. Liu, "MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System," *Proc. Int'l Conf. Extending Database Technology (EDBT),* 2004.

[8] B. Gedik, K.-L. Wu, P.S. Yu, and L. Liu, "Motion Adaptive Indexing for Moving Continual Queries over Moving Objects," *Proc. ACM Conf. Information and Knowledge Management (CIKM),* 2004.

[9] S. Ilarri, E. Mena, and A. Illarramendi, "A System Based on Mobile Agents for Tracking Objects in a Location-Dependent Query Processing Environment," *Proc. Int'l Workshop Database and Expert Systems Applications (DEXA),* pp. 577-581, 2001.

[10] C.S. Jensen, D. Lin, and B.C. Ooi, "Query and Update Efficient B+-Tree Based Indexing of Moving Objects," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 768-779, 2004.

[11] D.V. Kalashnikov, S. Prabhakar, S. Hambrusch, and W. Aref, "Efficient Evaluation of Continuous Range Queries on Moving Objects," *Proc. Database and Expert Systems Applications (DEXA),* pp. 731-740, 2002.

[12] G. Kollios, D. Gunopulos, and V.J. Tsotras, "On Indexing Mobile Objects," *Proc. ACM Principles of Database Systems (PODS),* pp. 261-272, 1999.

[13] I. Lazaridis, K. Porkaew, and S. Mehrotra, "Dynamic Queries over Mobile Objects," *Proc. Int'l Conf. Extending Database Technology (EDBT),* pp. 269-286, 2002.

[14] L. Liu, C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery," *IEEE Trans. Knowledge and Data Eng.*, pp. 610-628, 1999.
[15] M.F. Mokbel, X. Xiong, and W.G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *Proc. ACM Special Interest Group on Management of Data Conf. (SIGMOD)*, pp. 321-330, 2004.
[16] B.W. Parkinson, J.J. Spilker, P. Axelrad, and P. Eng, *Global Positioning System: Theory and Applications, Vol. 2.* Am. Inst. Aeronautics and Astronautics, 1996.
[17] J.M. Patel, Y. Chen, and V.P. Chakka, "Stripes: An Efficient index for Predicted Trajectories," *Proc. ACM Special Interest Group on Management of Data Conf. (SIGMOD)*, pp. 637-646, 2004.
[18] D. Pfoser, C.S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Object Trajectories," *Proc. Very Large Data Bases Conf. (VLDB)*, pp. 395-406, 2000.
[19] S. Prabhakar, Y. Xia, D.V. Kalashnikov, W.G. Aref, and S.E. Hambrusch, "Query Indexing and Velocity Constrained Indexing," *IEEE Trans. Computers*, vol. 51, no. 10, pp. 1124-1140, Oct. 2002.
[20] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez, "Indexing the Positions of Continuously Moving Objects," *Proc. ACM Special Interest Group on Management of Data Conf. (SIGMOD)*, 2000.
[21] Z. Song and N. Roussopoulos, "Hashing Moving Objects," *Proc. IEEE Mobile Data Management*, 2001.
[22] Z. Song and N. Roussopoulos, "*k*-Nearest Neighbor Search for Moving Query Point," *Proc. Symp. Spatial and Temporal Databases (SSTD)*, 2001.
[23] Y. Tao, D. Papadias, and Q. Shen, "Continuous Nearest Neighbor Search," *Proc. Int'l Conf. Very Large Data Bases*, 2002.
[24] Y. Tao, D. Papadias, and J. Sun, "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," *Proc. Very Large Data Bases Conf. (VLDB)*, 2003.
[25] D.B. Terry, D. Goldberg, D. Nichols, and B.M. Oki, "Continuous Queries over Append-Only Database," *Proc. ACM Special Interest Group on Management of Data Conf. (SIGMOD)*, 1992.
[26] Y. Theodoridis, E. Stefanakis, and T. Sellis, "Efficient Cost Models for Spatial Queries Using R-Trees," *IEEE Trans. Knowledge and Data Eng.*, vol. 12, no. 1, pp. 19-32, Jan./Feb. 2000.
[27] O. Wolfson, "Moving Objects Information Management: The Database Challenge," *Proc. Next Generation Information Technologies and Systems*, 2002.
[28] O. Wolfson, A.P. Sistla, S. Chamberlain, and Y. Yesha, "Updating and Querying Databases that Track Mobile Units," *Distributed and Parallel Databases*, vol. 7, no. 3, pp. 257-387, 1999.
[29] K.-L. Wu, S.-K. Chen, and P.S. Yu, "Processing Continual Range Queries over Moving Objects Using VCR-Based Query Indexes," *Proc. Int'l Conf. Mobile and Ubiquitous Systems (MobiQuitous)*, 2004.
[30] K.-L. Wu, S.-K. Chen, and P.S. Yu, "On Incremental Processing of Continual Range Queries for Location-Aware Services and Applications," *Proc. Int'l Conf. Mobile and Ubiquitous Systems (MobiQuitous)*, 2005.
[31] X. Xiong, M.F. Mokbel, and W.G. Aref, "SEA-CNN: Scalable Processing of Continuous k-Nearest Neighbor Queries in Spatio-Temporal Databases," *Proc. IEEE Int'l Conf. Data Eng.*, 2005.
[32] X. Yu, K.Q. Pu, and N. Koudas, "Monitoring K-Nearest Neighbor Queries over Moving Objects," *Proc. IEEE Int'l Conf. Data Eng.*, 2005.

Buğra Gedik received the BS degree from the Computer Engineering and Information Science Department at Bilkent University, Turkey. He is currently a PhD candidate in the College of Computing at the Georgia Institute of Technology. He conducts research on various aspects of distributed data intensive systems, including peer-to-peer computing, mobile data management and location-based services, and sensor network computing. His research emphasis is on developing system-level architectures and techniques to address scalability problems in distributed information monitoring services. He was a program committee member for the 22nd IEEE International Conference on Data Engineering (ICDE 2006) and received the best paper award at the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS 2003). He is a student member of the IEEE Computer Society.

Kun-Lung Wu received the BS degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, and the MS and PhD degreess in computer science from the University of Illinois at Urbana-Champaign. He is with the IBM Thomas J. Watson Research Center, currently as a member of the Software Tools and Techniques Group. His recent research interests include data streams, continual queries, mobile computing, Internet technologies and applications, database systems, and distributed computing. He has published extensively and hold many patents in these areas. He is a senior member of the IEEE Computer Society and a member of the ACM. He was an associate editor for the *IEEE Transactions on Knowledge and Data Engineering*, 2000-2004. He was the general chair for the Third International Workshop on E-Commerce and Web-Based Information Systems (WECWIS 2001). He received a best paper award from IEEE EEE 2004. He is an IBM Master Inventor.

Philip S. Yu received the BS degree in electrical engineering from National Taiwan University, the MS and PhD degrees in electrical engineering from Stanford University, and the MBA degree from New York University. He is with the IBM Thomas J. Watson Research Center and is currently the manager of the Software Tools and Techniques Group. His research interests include data mining, Internet applications and technologies, database systems, multimedia systems, parallel and distributed processing, and performance modeling. He has published more than 430 papers in refereed journals and conferences and holds or has applied for more than 250 US patents. He is a fellow of the ACM and a fellow of the IEEE. He is associate editor of *ACM Transactions on the Internet Technology* and *The ACM Transactions on Knowledge Discovery in Data*. He is a member of the IEEE Data Engineering steering committee and is also on the steering committee of the IEEE Conference on Data Mining. He was the editor-in-chief of *IEEE Transactions on Knowledge and Data Engineering* (2001-2004) and also served as an associate editor of *Knowledge and Information Systems*. He will serve as the general chair of the 2006 ACM Conference on Information and Knowledge Management and the program chair of the 2006 joint conferences of the Eighth IEEE Conference on E-Commerce Technology (CEC' 06) and the Third IEEE Conference on Enterprise Computing, E-Commerce and E-Services (EEE '06). He served as the general chair of the 14th IEEE Intl. Conference on Data Engineering and the general co-chair of the Second IEEE International Conference on Data Mining. He received an Outstanding Contributions Award from IEEE International Conference on Data Mining in 2003 and also an IEEE Region 1 Award for "promoting and perpetuating numerous new electrical engineering concepts" in 1999. He is an IBM Master Inventor.

Ling Liu is an associate professor at the College of Computing at Georgia Institute of Technology. There, she directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining research issues and technical challenges in building large-scale distributed computing systems that can grow without limits. Their work has included peer-to-peer computing, data-grid computing, and enterprise computing systems. She has published more than 150 international journal and conference articles. Her research group has produced a number of software systems that are either open source or directly accessible online, among which the most popular ones are WebCQ and XWRAPElite. Most of Dr. Liu's current research projects are sponsored by NSF, DoE, DARPA, IBM, and HP. She is on the editorial board of several international journals, such as *IEEE Transactions on Knowledge and Data Engineering*, *International Journal of Very Large Database Systems (VLDBJ)*, and *International Journal of Web Services Research*. She has chaired a number of conferences as a program committee chair, a vice program committee chair, or a general chair. She is a member of the IEEE