RC23302 (W0408-099) August 23, 2004
Computer Science

IBM Research Report

Motion Adaptive Indexing for Moving Continual Queries over Moving Objects

Bugra Gedik
College of Computing
Georgia Institute of Technology

Kun-Lung Wu, Philip S. Yu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Ling Liu
College of Computing
Georgia Institute of Technology

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Motion Adaptive Indexing for Moving Continual Queries over Moving Objects

Buğra Gedik
College of Computing
Georgia Institute of Technology
bgedik@cc.gatech.edu

Kun-Lung Wu, Philip S. Yu
T.J. Watson Research Center
Hawthorne, NY 10532
{*klwu, psyu*}*@us.ibm.com*

Ling Liu
College of Computing
Georgia Institute of Technology
lingliu@cc.gatech.edu

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* 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 motion sensitive bounding boxes, where updates to the bounding boxes are needed only when objects move across the boundaries. 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. Our experiments show that the proposed motion adaptive indexing scheme is efficient for the evaluation of moving continual *range* queries.

1 Introduction

With the continued advances in mobile computing and positioning technologies, such as GPS [11], 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*.

MCQs have different applications such as environmental awareness, object tracking and monitoring, and location-based services. Here is an example moving continual query MCQ_1: “Give me the positions of those customers who are looking for taxi and are within 5 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 marching on the road. The focal object of MCQ_1 is the taxi on the road.

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 still objects in the query region. An example of such a query is MCQ_2: "Give me the locations and names of the gas stations offering gasoline for less than $1.2 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 gas stations within 10 miles with respect to the location of the car. Another interesting specialization is so called *static continual query over moving objects*, where the queries are posed with static focal objects or without focal objects. An example query is MCQ_3: "Give me the list of AAA vehicles that are currently on service call in downtown Atlanta (or 5 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 like MCQ_1.

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 [13, 7, 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 [5]; the spatial region of the query moves continuously as the query's focal object moves. Moving continual queries introduce a new challenge in indexing, mainly due to the highly dynamic nature of both queries and objects.

Due to frequent updates to the index structures, traditional indexing approaches built on moving object positions do not work well [13, 7]. In order to tackle this problem, several researchers have introduced alternative approaches based on the idea of indexing the parameters of the motion functions of the moving objects [8, 14, 18, 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 mostly are based on R-tree-like structures and produce time parameterized minimum bounding rectangles that enlarge continuously [14, 18, 13]. As a consequence of enlarged bounding rectangles, the search performance can deteriorate over time and the index structures may need to be reconstructed periodically [13, 14]. As far as update costs are concerned, approaches based on time parameterized rectangles [14, 18] can provide excellent performance. However, they are not *sufficient* for processing MCQs. This is because they do not support incremental re-evaluation of queries and the **continual** nature of these queries dictates that the

same queries must be re-evaluated at frequent intervals. Thus, a need is recognized to have a new method 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 MSBs, 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 MSBs is to facilitate incremental processing of MCQs. We provide two techniques to reduce the costs of query re-evaluation 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. MSBs 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 MSBs to the changing moving 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.

The proposed motion-adaptive indexing scheme is independent of the underlying spatial index structures by design. Any spatial index method, such as an R-tree, can be used to index the MSBs.

The MAI approach can also be extended to the evaluation of moving continual *k-nearest neighbor* queries. However, due to space limitation, we focus on moving continual range queries in this paper. In [6], the concepts of *guaranteed safe radius* and *optimistic safe radius* were introduced to extend MAI for evaluating moving continual kNN queries.

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* (kNN) queries [6]. 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.

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. A recent study dealing with the problem of indexing and querying moving object trajectories

can be found in [12]. Continual queries are used as a useful tool for monitoring frequently changing information [19, 10]. In the spatial databases domain, continual queries are employed for continually querying moving object positions. Most of the work on continual queries over moving object positions is either on static continual queries over moving objects [13, 7, 8, 3, 15, 23, 24] or on moving continual queries over static objects [17, 16].

In [13], velocity constrained indexing and query indexing (Q-index) has been proposed for efficient evaluation of static continual range queries. The same problem is studied in [7], however the focus is on in-memory structures and algorithms. In [14], 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 future (predictive), is described in [18]. In [2], efficient query evaluation techniques for nearest neighbor ($k = 1$) and reverse nearest neighbor queries are developed for moving queries over moving objects. Work on moving continual queries over static objects focuses on continuous k-nearest neighbor (CNN) evaluation. An algorithm for precalculating k-nearest neighbors with a line segment representing the continuous motion of an object, is described in [17]. Note that even though TPR-related indexes [14, 18, 2] support moving queries, these moving queries are predefined regions in the spatial-temporal domain. They are not the moving continual queries discussed in this paper. None of them has addressed the problem of moving continual queries over moving objects.

The concept of moving continual queries is to some extent similar to Dynamic Queries (DQ) [9]. A dynamic query is defined as a temporally ordered set of snapshot queries in [9]. This is a low level definition as opposed to our definition of moving continual queries which is more declarative and is defined from users' perspective. The work done in [9] indexes the trajectories of the moving objects and describes how to efficiently evaluate dynamic queries that represent predictable or non-predictable 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 dynamic motion behavior of the objects.

In [15], 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

	Query Types			System Properties			
	Moving Q Static O	Static Q Moving O	Moving Q Moving O	Incremental Evaluation	Predictive Query Results	Index Independence	Motion Adaptation
MAI	•	•	•	•	•	•	•
DQ [9]	•		•	•			
CNN [17]	•				◦ [1]		
Q-index [13]		•		•		•	
TPR [14]		•	• [2]				

Table 1: Comparison of motion-adaptive index with existing approaches

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 has to be propagated to location preprocessors that has the exact information. In [3] and [5], 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 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

3.1 Basic Concepts

We denote the set of moving or still 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 still 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: $\langle i_o, \vec{p}, \vec{v}, a_p \rangle$. Here, i_o is the unique object identifier, $\vec{p} = (p_x, p_y)$ 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, $\vec{v} = (v_x, v_y)$ is the current velocity vector of the object, and a_p is a set of properties about the object. A still object can be modeled as a special case of moving objects where the velocity vector is set to zero, $\forall o_s \in O_s, o_s.\vec{v} = (0, 0)$.

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

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

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 queries either have no focal objects or the focal object is a still object. Namely, $\forall q_s \in Q_s, q_s.i_o = null \vee q_s.i_o \in O_s$.

3.2 Motion Modeling and Update

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

Motion Modeling – Modeling motions of the moving objects for predicting their positions is a commonly used method in moving object indexing [21, 8]. 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. In order to evaluate moving queries in 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 $\vec{v}$, its position is $\vec{p}$ 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 $\vec{p} + \Delta t * \vec{v}$.

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. We below 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 its position as predicted 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. The tradeoffs between the inaccuracy introduced due to motion modeling and the savings in terms of number of updates generated, is controlled by the parameter ΔD. However, we do not discuss considerations with respect to the setting of ΔD in this paper. An

extensive study of motion update policies and their tradeoffs is given in [22].

4 Efficient Evaluation of Range MCQs

In this section we describe the motion adaptive indexing scheme for efficient processing of moving range queries over moving objects.

4.1 Motion Sensitive Bounding Boxes

Motion sensitive bounding boxes (MSBs) 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, 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 Figure 1 for illustrations).

Let $Rect(l, m)$ denote a rectangle with l and m as any two end points of the rectangle that are on the same diagonal. Let $sign(\vec{x})$ denote a function over a vector $\vec{x}$, which replaces each entry in $\vec{x}$ with its sign (+1 or -1). Then we define the MSB for a moving object o and the MSB for a moving query q with focal object o_f as follows:

$$\forall o \in O_m, MSB(o) = Rect(o.\vec{p}, o.\vec{p} + \alpha(o) * o.\vec{v})$$

$$\forall q \in Q_m, MSB(q) = Rect(o_f.\vec{p} - q.r * sign(q.\vec{v}),$$
$$o_f.\vec{p} + \beta(q) * q.\vec{v} + q.r * sign(q.\vec{v}))$$

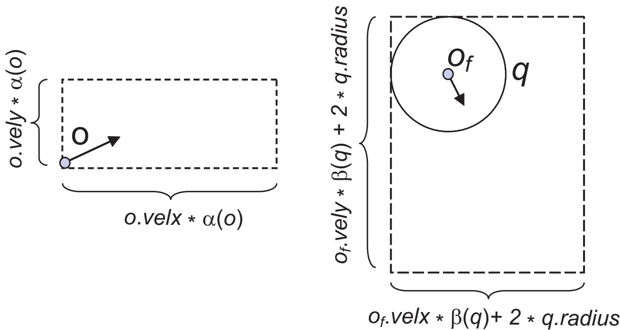

Figure 1: MSBs

For each moving query, its MSB is calculated and used in place of the query's spatial region in the query-based MSB index, referred to as the $Index_q^{msb}$. Similarly, for each moving object, its MSB is calculated and used in place of the object's position and we refer to such an object-based MSB index as the $Index_o^{msb}$.

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 posi-

tion 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 crucial to note that, using MSBs 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.* 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 [14, 13], motion sensitive bounding boxes require almost no significant change to the underlying spatial index implementation. (2) They form a natural 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).

4.2 Predictive Query Results on Per Object Base

It is well known that one way of saving IO and improving efficiency of evaluating moving queries is to precalculate future results of the continual queries. This approach has been successfully used in the context of continual moving kNN queries over *static* objects [17]. Most of existing approaches to precalculating query results associate a time interval to 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 anyone 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 straight forward 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 Figure 2).

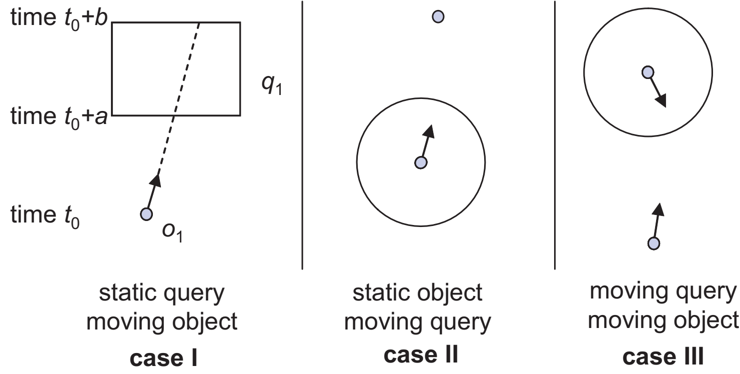

Figure 2: Calculating Intervals

Similarly, to calculate the intersection point of a moving query and a moving or non-moving 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 $o_f \in O_m$, and $o \in O$ 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: $Dist(o_f.\vec{p} + t * o_f.\vec{v}, o.\vec{p} + t * o.\vec{v}) \leq q_m.r$. Figure 2 illustrates three different cases that arise in the calculation of prediction time interval for each per-object based predictive query result entry.

The predictive query results are precalculated on per object basis and predictive query 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 to 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 require to be refreshed when the objects in the predictive query results have moved away from the proximity of the query or when

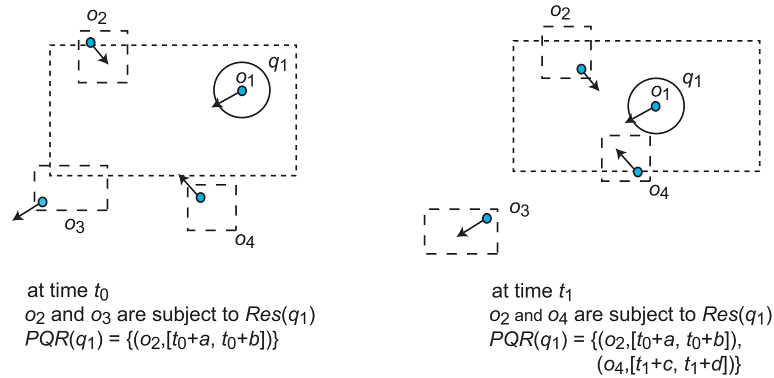

Figure 3: An illustration of how PQRs integrate with MSBs

the objects that did not participate in the prediction have entered the proximity of the query.

4.3 Determining PQRs Using MSBs

MSBs are used to effectively determine for which objects we should perform result prediction with respect to a query. Concretely, for a given query, objects whose MSBs intersect with the query's MSB are considered as potential candidates of the query's predictive result. Figure 3 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 Figure 3. 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 MSBs. At time t_0 only objects o_2 and o_3 are subject to query q_1's PQR, as their MSBs 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 MSBs. However objects o_3 and o_4 have changed their MSBs. 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 non-empty 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.

4.4 Motion Adaptive Indexing

In this subsection, 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.

4.4.1 Processing Moving Queries: An Overview

The evaluation of moving queries is performed through multiple query evaluation steps 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 MSBs of the objects accompanied by the associated motion functions as data. Static objects have point MSBs. Similarly, $Index_q^{msb}$ stores the MSBs of the queries accompanied by the associated motion functions of the focal objects of the queries and their radiuses as data. Static queries have MSBs 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 step 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.

Moving Object Table (MOT): An MOT entry is described as $(i_o, i_q, \vec{p}, \vec{v}, t, B_{msb}, P_{cm}, V_{ch})$ 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, $\vec{p}$ is the last received position, $\vec{v}$ is the last received velocity vector of the moving object, t is the timestamp of the motion updates ($\vec{p}$ and $\vec{v}$) 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 (described later in Section 4.5) of the object and V_{ch} is a Boolean variable indicating whether the object has changed its motion function since the last query evaluation step.

Moving Query Table (MQT): An MQT entry is described as $(i_q, \vec{p}, \vec{v}, r, t, B_{msb}, P_{cm}, V_{ch})$ and stores information regarding a moving query. Here, i_q is the moving query identifier, $\vec{p}$ and $\vec{v}$ are the last received position and the last received velocity vector of the query's focal object respectively, t is the timestamp of the motion updates ($\vec{p}$ and $\vec{v}$) 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 on the period of constant motion (described later in Section 4.5) of the object and V_{ch} is a Boolean variable indicating whether the focal object has changed its motion function since the last query evaluation step or not. 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 will need to look

them up from the moving object table, which is quite costly.

The MOT and MQT table entries are updated whenever new motion updates are received from the moving objects. The effect of a motion update is reflected on the query results when the next periodic query evaluation step is performed. Assuming that moving objects decide whether they should send new motion updates or not at every P_{mu} seconds (called the *motion update time period*), one of our aims is to perform a single query evaluation step in less that P_{mu} seconds in order to provide fresh query results, i.e. having $P_s \leq P_{mu}$. At each query evaluation step, we need to perform *query table scan* and *object table scan*. The scan algorithms presented in the next subsection describe how these two tasks are performed.

4.4.2 The Scan Algorithms

At each query evaluation step, 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 $Index_o^{msb}$.

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 $\vec{p}, \vec{v}, t$, and B_{msb}) or changed its motion function since the last query evaluation period (based on V_{ch}). 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 $|\vec{v}|$ and P_{cm} (See Section 4.5 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 MSBs 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 MSBs intersect with the new MSB of the object are retrieved. For all those queries, result prediction is performed against the object. Lastly, the query result entries obtained from the prediction with non-empty 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 $\vec{p}, \vec{v}, r, t$, and B_{msb}) or its focal object

has changed its motion function since the last query evaluation step (based on V_{ch}). 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 $|\vec{v}|$ and P_{cm} (See Section 4.5 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 MSBs 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 non-empty 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 MSBs are not invalidated and their focal objects have not changed their motion function. For queries that have invalidated their MSBs 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.

4.5 Adaptive Parameter Selection

The α and β parameters used for calculating MSBs 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: (a) *the speed of the object* and (b) *the period of constant motion of the object*, P_{cm} (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 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. The P_{cm} entries in the MOT and MQT tables, that are used for the purpose of adaptive parameter selection, are updated using a simple weighted running average based on the interarrival times of the position updates received from the moving objects.

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 step. This model is used as the guide to build an off-line

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

Table 2: Symbols and their meanings

computed $\alpha\beta$Table, giving the best α and β values for different value pairs of speed and period of constant movement of a moving object. Table 2 lists some of the symbols used and their meanings.

Let A_{mo} denote the average area of a moving object motion sensitive bounding box and A_{mq} denote the average area of a moving query motion sensitive bounding box. Then, assuming that the x and y components of the velocity vector are equal, based on the definition of MSBs, we have:

$$A_{mo} \approx (\alpha * |\vec{v}|/\sqrt{2})^2 \text{ and } A_{mq} \approx (\beta * |\vec{v}|/\sqrt{2} + 2 * R_{mq})^2$$

Let A_o denote the average size of the object bounding boxes stored in the $Index_o^{msb}$ (static object's 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:

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

Given this information, the following four quantities can be analytically derived based on well studied R-tree cost models [20]: 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; (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 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,

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

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. Then, we have:

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

Let N_{mot} denote the expected value of the number of entries in the object table that requires processing and N_{mqt} denote the expected value of the number of entries in the query table that requires processing. Assuming that an object causes a velocity change event independent of whether it has caused an MSB invalidation event and similarly a query focal object causes a velocity change event independent of whether the query has caused an MSB invalidation, we have:

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

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, 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, an update on the $Index_q^{msb}$ and a search on the $Index_o^{msb}$ are needed, as follows:

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

4.5.2 Building and Using the $\alpha\beta Table$

The cost function developed in this section has a global minimum that optimizes the IO cost of the query evaluation. We build an off-line computed $\alpha\beta Table$, which gives the optimal α and β values for different value pairs of object speed ($\vec{v}$) 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 columns correspond to different periods of constant motion and the entries are optimal (α, β) pairs. Recall 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

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 time period	30 seconds

Table 3: System Parameters

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 off-line computed $\alpha\beta Table$. We provide results on the effect of adaptive parameter selection in Section 5.3.

5 Experimental Results

This section describes five sets of implementation based 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. 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 number 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.

5.1 System Parameters and Setup

In the experiments presented in the rest of the paper, the parameters take their default values listed in Table 3, when not specified otherwise. Based on the default values, 50% of the objects are moving and

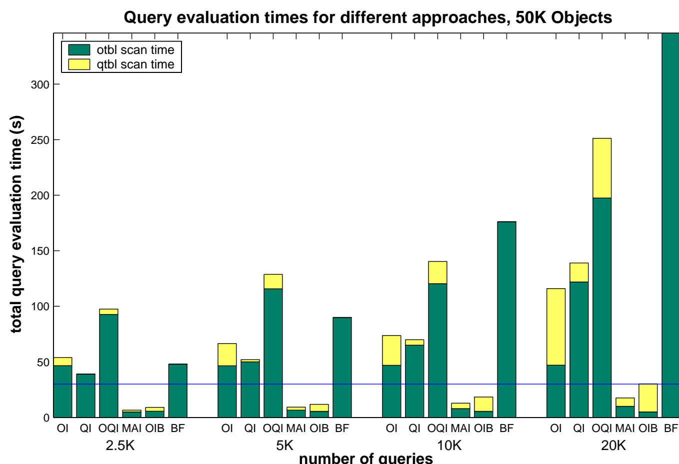

Figure 4: Query evaluation time

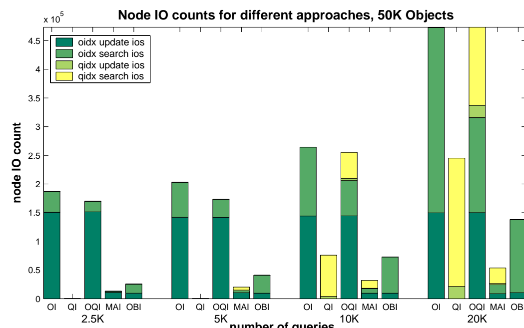

Figure 5: Query evaluation node IO

the remaining 50% are static. Similarly, 50% of the queries are moving and the remaining 50% are static. 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 query 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 [13, 14]. Objects and queries are randomly distributed in the area of interest, except in Section 5.4 where we consider skewed distributions. Objects that belong to different classes with strictly varying movement behaviors are considered in Section 5.3. 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 3 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 4KB 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 5.4 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 when objects move [3] and searched periodically in order to evaluate the query results. The QI approach uses a query index that is updated when queries move and searched when new object positions are received in order to update the query results incrementally. OQI is a stripped down version of MAI without MSBs and PQRs. OIBs 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 PQRs).

5.2 Performance Comparison

Figure 4 plots the total query evaluation time for fixed number of objects (50K) with varying number of queries (2.5K to 20K). 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 time period P_{mu} in this set of experiments. Figure 5 plots the query evaluation node IO count for the same setup. The node IO is divided into four different components. These are: (a) node IO due to object index update, (b) node IO due to object index search, (c) node IO due to query index update and (d) node IO due to query index search. Each component is depicted with a different color in Figure 5. Several observations can be obtained from Figure 4 and Figure 5.

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 2.5K queries, as shown in Figure 4. 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 object index update component of the OI in Figure 5. 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 20K queries in Figure 5). 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 [14, 18]), MAI still performs significantly better than such an object index based approach.*

Second, the approaches with a query index that is searched for large number of objects, do not perform well for large number of queries. This is clear from the poor performances of QI and OQI for 20K queries, as shown in Figure 4. 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 Figure 5. Note that, for small number of queries node IO count for QI appears as 0, because the query

[3] Although update efficient object indexes exist [14, 18], we show that their use does not change our conclusions for large or moderate number of queries, in which case search cost dominates

index fits into the LRU buffer.

Third, the brute force approach performs relatively good compared to OQI and slightly better compared to OI, when the number of queries is small (2.5K), as shown in Figure 4. 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% savings in query evaluation time under all cases when compared to the best competing approach except OIB. OIB performs reasonably well, but fails to scale well with increasing number of queries when compared to the proposed MAI approach.

5.3 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 (avg. period of constant motion 1 minute) and move slow (max. speed 20 miles/hour). The second class of moving objects possess the default properties described in Section 5.1. The third class of moving objects seldom change their motion functions (avg. period of constant motion 30mins) and move fast (max. 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 non-adaptive case.

Figure 6 plots the time and IO cost of query evaluation for MAI and 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 4 times the number of objects belonging to the second class. Dually, 1:4:1 represents the case where the second class cardinality is 4 times the other two class cardinalities. 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 Figure 6.

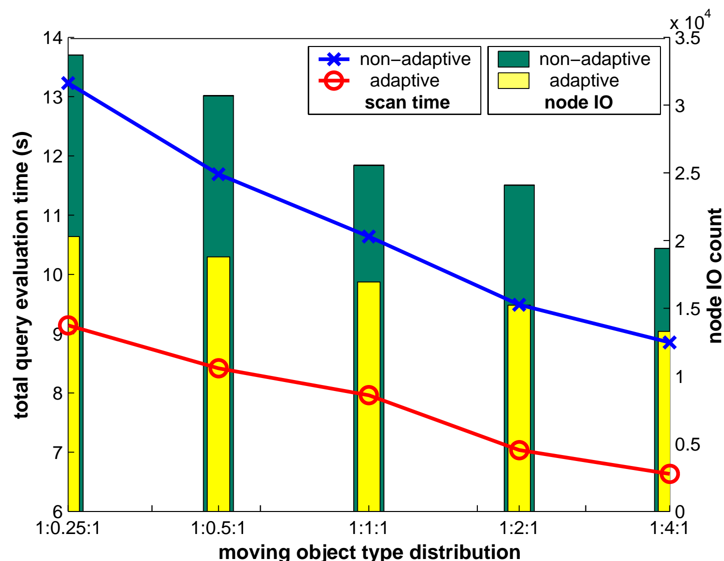

Figure 6: Performance gain due to adaptive parameter selection

First, we notice that the adaptive parameter selection has a clear performance advantage. This is clearly observed from Figure 6, 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.

5.4 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 pick N_h different positions within the area of interest randomly, 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.

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 [4]. 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 [7] 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.

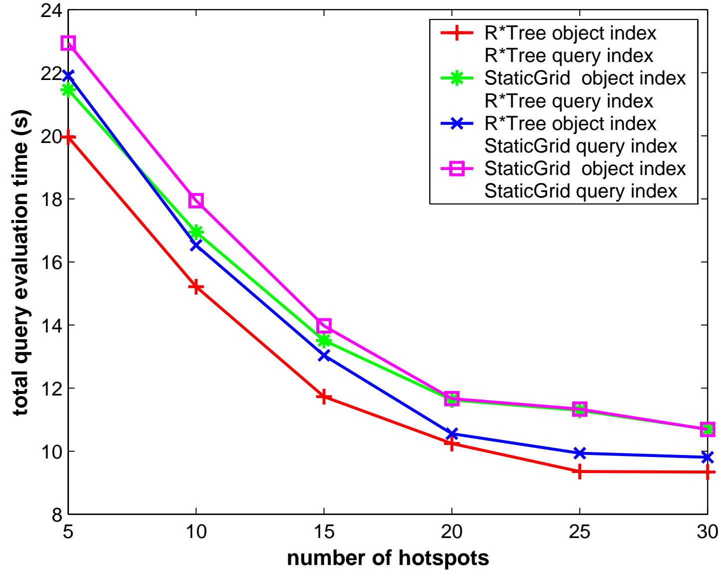

Figure 7: Effect of data and query skewness on performance

Figure 7 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. Figure 7 shows that decreasing the number of hot spots exponentially increases the query evaluation scan time. But even for $N_h = 5$, the query evaluation time does not exceed the query evaluation period. Figure 7 also shows that R*-tree performs the best under all conditions.

5.5 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 5.1 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.

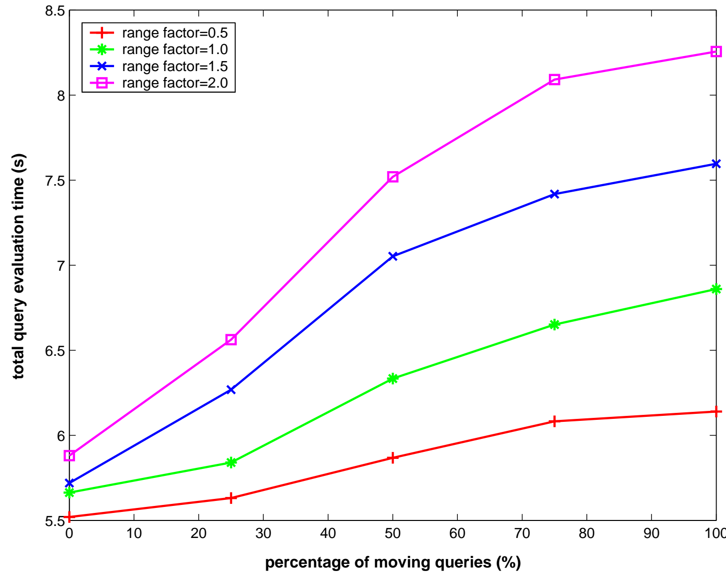

Figure 8: Effect of query range and moving query percentage on performance

Figure 8 plots the total query evaluation scan time as a function of moving query percentage for different range factors. As shown in Figure 8, 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 on the query evaluation time.

In Figure 9 we study the effect of the number of objects on the query evaluation performance. Figure 9 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 5K. From Figure 9 we observe a linear increase in scan time with the increasing number of objects, where the R*-tree implementation of $Index_o^{msb}$ and $Index_q^{msb}$ show better scalability with increasing number of objects than the static grid implementation for the similar reason discussed before.

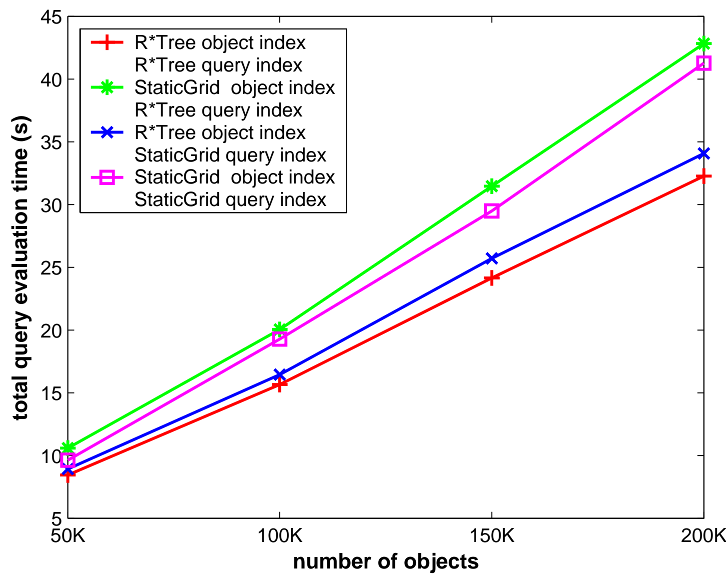

Figure 9: Effect of number of objects on performance

6 Conclusion

We presented a system and a motion-adaptive indexing scheme for efficient processing of moving continual queries over moving objects. We reported a series of experimental performance results for different workloads and demonstrated the effectiveness of our motion adaptive indexing scheme through comparisons with other alternative indexing mechanisms.

References

[1] C. C. Aggarwal and D. Agrawal. On nearest neighbor indexing of nonlinear trajectories. In *ACM PODS*, 2003.

[2] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, 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. In *IEEE IPCCC*, 2002.

[4] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[5] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, 2004.

[6] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Processing moving queries over moving objects using motion adaptive indexes. Technical Report CERCS-04-06, Georgia Tech., 2004.

[7] D. V. Kalashnikov, S. Prabhakar, S. Hambrusch, and W. Aref. Efficient evaluation of continuous range queries on moving objects. In *DEXA*, 2002.

[8] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *ACM PODS*, 1999.

[9] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *EDBT*, 2002.

[10] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, pages 610–628, 1999.

[11] B. W. Parkinson, J. J. Spilker, P. Axelrad, and P.Eng. *Global Positioning System: Theory and Applications*, volume 2. American Institute of Aeronautics and Astronautics, 1996.

[12] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, 2000.

[13] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[14] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD*, 2000.

[15] Z. Song and N. Roussopoulos. Hashing moving objects. In *IEEE Mobile Data Management*, 2001.

[16] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, 2001.

[17] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, 2002.

[18] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, 2003.

[19] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only database. In *ACM SIGMOD*, 1992.

[20] Y. Theodoridis, E. Stefanakis, and T. Sellis. Efficient cost models for spatial queries using R-trees. *IEEE TKDE*, 12(1):19–32, 2000.

[21] O. Wolfson. Moving objects information management: The database challenge. In *Next Generation Information Technologies and Systems*, 2002.

[22] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[23] K.-L. Wu, S.-K. Chen, and P. S. Yu. Indexing continual range queries for location-aware mobile services. In *IEEE International Conference on e-Technology, e-Commerce, and e-Service*, 2004.

[24] K.-L. Wu, S.-K. Chen, and P. S. Yu. Processing continual range queries over moving objects using VCR-based query indexes. In *ACM Mobiquitos*, 2004.