Processing Moving Queries over Moving Objects Using Motion Adaptive Indexes

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

Kun-Lung Wu, Philip S. Yu
T.J. Watson Research Center
IBM Research
{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 queries (MQs) over moving objects. It uses the concept of *motion-sensitive bounding boxes* to model the dynamic behavior of both moving objects and moving queries. Instead of indexing frequently changing object positions, we index less frequently changing motion sensitive bounding boxes together with the motion functions of the objects. This significantly decreases the number of update operations performed on the indexes. We use *predictive query results* to optimistically precalculate query results, thus decreasing the number of search operations performed on the indexes. More importantly, we propose a *motion adaptive* indexing method. Instead of using fixed parameters for motion sensitive bounding boxes, we automatically adapt the sizes of the motion sensitive bounding boxes to the dynamic motion behaviors of the corresponding individual objects. As a result, the moving queries can be evaluated faster by performing fewer IOs. Furthermore, we introduce the concept 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 evaluation of both moving continual *range* queries and moving continual kNN queries.

1 Introduction

With the emergence of positioning technologies like GPS [16], and the continued advances of mobile computing industry, 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. *Moving queries over moving objects* (MQs for short) are moving continual spatial queries over moving object positions. Efficient evaluation of moving queries over moving objects is an important issue in both mobile systems and moving object tracking systems. There are two major types of MQs − *moving range queries* and *moving k-Nearest Neighbor queries* (kNN).

Research on evaluating range queries over moving object positions has so far focused on static continual range queries [18, 9, 19, 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 range query exhibits some fundamental difference when compared to a static range query. A moving range query has an associated moving object, called the *focal object* of the query [8]; the spatial region of the query moves continuously as the query's focal object moves. Moving queries introduce a new challenge in indexing, mainly due to the highly dynamic nature of the system elements.

MQs 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 moving query MQ_1: "Give me the positions of those customers who are looking for taxi and are within 5 miles (of my location at each instance 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 MQ_1 is the taxi on the road.

Different specializations of MQs can result in interesting and useful classes of MQs. One is called *moving queries over static objects*, where the target objects are still objects in the query region. An example of such a query is MQ_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 buildings within 10 miles with respect to the location of the car on the move. Another interesting specialization is so called *static query over moving objects*, where the queries are posed with static focal objects or without focal objects. An example query is MQ_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".

Traditional indexing mechanisms do not work well for indexing moving object positions due to the frequent updates required on the index structures [18, 9]. In order to tackle this problem, recently several researchers have introduced indexing mechanisms based on indexing the parameters of the motion functions of the moving objects [11, 19, 1]. This approach alleviates the problem of frequent updates to the index, as the index needs to be updated only when the motion function of an object changes. However this kind of indexing, mostly based on R-tree like structures, produces time parameterized minimum bounding rectangles that enlarge continuously and result in the deterioration of the search performance over time. As a result, most of these index structures are used for a certain interval after which they have to be reconstructed [18, 19].

In this paper, we describe a motion-adaptive indexing scheme for efficient processing of moving queries over moving objects. It uses the concept of *motion-sensitive bounding boxes* to model the dynamic behavior of both moving objects and moving queries. Instead of indexing frequently changing object positions, we index less frequently changing motion sensitive bounding boxes together with the motion functions of the objects. This significantly decreases the number of update operations performed on the indexes. We provide two techniques to address the problem of increased

search cost on such indexes. First, we optimistically precalculate query results and maintain such *predictive query results* under the presence of object motion changes. Second, we support *motion adaptive* indexing. Instead of using fixed size of motion sensitive bounding boxes, we automatically adapt the sizes of the motion sensitive bounding boxes to the changing movement 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 introduce the concept of guaranteed safe radius and optimistic safe radius to extend our motion adaptive indexing scheme to the evaluation of moving continual *k-nearest neighbor* queries.

Our experimental results show that the motion adaptive indexing scheme is efficient for evaluation of both moving continual *range* queries and moving continual *k-nearest neighbor* (kNN) queries. 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. 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. It should be mentioned that the moving queries, as we formalize them in this paper, are different from moving queries with a predefined path [19]. To some extent, our moving query definition is close to dynamic queries in [12]. There are, however, some important differences. Dynamic queries are defined as temporally ordered set of snapshot queries. Instead of using a procedural level definition, the concept of moving queries is defined declaratively from a users perspective [8].

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

2 The System Model

The basic elements of our system model are a set of moving or still 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 re-evaluation, but also increases the scalability of the system by enabling timely evaluation of large number of moving queries over large number of objects.

2.1 Basic Concepts and Problem Statement

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 range queries and Q_s denotes the set of static range queries.

Moving Objects. We describe a moving object $o_m \in O_m$ by a quadruple: $\langle oid, pos, vel, \{props\}\rangle$. Here, oid is the unique object identifier, pos is the current position of the moving object, $vel = (velx, vely)$ is the current velocity vector of the object, where $velx$ is its velocity in the x-dimension and $vely$ is its velocity in the y-dimension, and $\{props\}$ 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, $\forall o_s \in O_s, o_s.vel = (0, 0)$.

Moving Queries. We describe a moving query $q_m \in Q_m$ by a quadruple: $\langle qid, oid, region, filter\rangle$. Here, qid is the unique query identifier, oid is the object identifier of the focal object of the query, $region$ defines the shape of the spatial query region bound to the focal object of the query, and $filter$ is a Boolean predicate defined over the properties ($\{props\}$) of the target objects of the query. Note that, $region$ 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. 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. I.e., $\forall q_s \in Q_s, q_s.oid = null \vee q_s.oid \in O_s$.

Before we describe the motion modeling basics, we first review three basic types of indexing techniques for evaluating range queries over moving objects and discuss their advantages and inherent weaknesses. The three indexing mechanisms we consider are *Object-only Indexing* (OI), *Query-only Indexing* (QI) and *Object and Query Indexing* (OQI). To simplify the discussion without loss of generality, we consider these indexes in their simplest forms and we assume that we are fed with a stream of object position updates, where an update is received at each time step from every object that has moved since the last time step. We comment on whether a particular indexing approach is open to optimizations but will not consider specific optimizations in the discussion.

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 step, all queries are evaluated against the object index. An inherent drawback of the basic object-only indexing approach is the re-evaluation of all queries against the object index regardless of whether we have a static or moving query and whether the object position changes are of interest to the query or not. 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 [18] and time parameterized R-trees in [19]).

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 step, 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

step. 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 [18].

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 step, 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 step, each *new* query position is evaluated against the object index and the new result of the query is determined. The object and query indexing approach evaluates object positions against the query index only for the objects that have changed their positions since the last query evaluation step as opposed to query-only indexing which has to do it for all objects. It also evaluates the queries against the object index only for queries that have moved since the last query evaluation step, as opposed to object-only indexing which has to do it for all queries. Although the object and query indexing approach incurs higher cost due to maintenance of an additional index structure, it is possible to employ a larger range of optimizations to reduce the additional cost incurred and it does not have certain restrictions of object-only indexing or query-only indexing.

2.2 Overview of the Solution Approach

Bearing in mind 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. A motion adaptive index is defined as an index of parameterized motion-sensitive bounding boxes. 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 together with the motion functions of the objects. 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 $Index_q^{msb}$).

To address the problem of increased search cost due to querying both $Index_o^{msb}$ and $Index_q^{msb}$, we employ two optimization techniques: the *predictive query results*, which optimistically precomputes the query results in the presence of object motion changes, and the *motion adaptive* indexing, which dynamically adapts the sizes of the motion sensitive bounding boxes to the changing motion behaviors at the granularity of individual objects, allowing moving queries to be evaluated faster by performing fewer IOs. 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*.

Motion Modeling

Modeling motions of the moving objects for predicting their positions is a commonly used method in moving object indexing [26, 11]. 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 [1]. 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 vel, its position is pos 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 $pos + \Delta t * vel$. We use a linear motion function in this paper, since it is the commonly used model in moving object databases. We refer readers to [1] for a study of non-linear 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. 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.

3 Efficient Evaluation of Moving Range Queries

3.1 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 $sign(x)$ denote a function over a vector x, which replaces each entry in 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:

[1] This technique is known as dead reckoning [6].

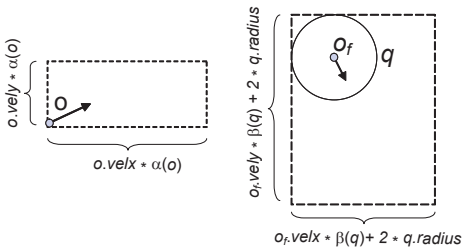

Figure 1: Motion sensitive bounding boxes

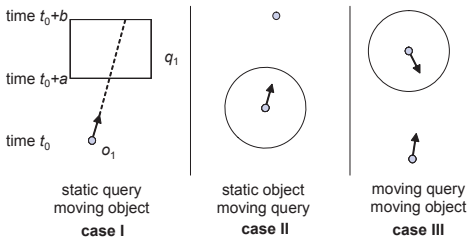

Figure 2: Calculating Intervals

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

For each moving query, its MSB is calculated and used in place of the query's spatial region in the query-based MSB index, 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 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 significantly 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 $Index_q^{msb}$).

Although maintaining MSB indexes increase the cost of searching the index due to higher overlap of spatial objects (MBRs) being indexed, it is important to note that for appropriate values of the α and β parameters, the gain of using MSB indexes is significant. Therefore, we need not only mechanisms for reducing the cost of search operations in the MSB indexes, but also 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. It is also crucial to note that, using MSBs does not introduce any inaccuracy in the query results, as we store the motion function of the object or the query together with its MSB inside the spatial index[2]. Furthermore, MSBs provide the following two advantages: (1) As opposed to approaches that alter the implementation of traditional spatial indexes for decreasing the update cost [19, 18], motion sensitive bounding boxes require almost no significant change to the underlying spatial index implementation. (2) Motion sensitive bounding boxes perform size adaptation at the granularity of individual objects, leading to significant reduction of IO cost (See Section 3.5 for further detail).

3.2 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 continuous moving kNN queries over static objects [21]. 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 base.

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). 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 Euclidian 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: $Dist(o_f.pos + t * o_f.vel, o.pos + t * o.vel) \leq q_m.radius$. 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 base 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 moving queries:

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,

[2]The inaccuracy due to motion modeling is not considered here. See [27] for a discussion of motion update policies and their tradeoffs.

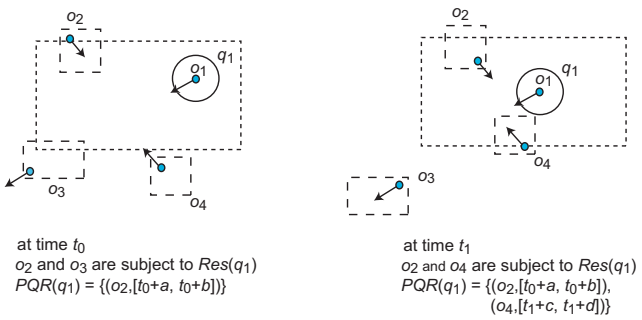

Figure 3: An illustration of temporal query results and how it integrates with motion sensitive bounding boxes

as the predicted results are less likely to hold until those objects reach to the proximity of the query, e.g., entering the motion sensitive bounding box 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 the objects that did not participate in the prediction have entered the proximity of the query.

3.3 Determining Predictive Query Results Using MSBs

In addition to serving as an optimization technique to minimize the update cost on the spatial indexes, MSBs can be used to effectively determine for which objects we should perform result prediction with respect to a query (answering the first question listed in Section 3.2). 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 predictive query result, 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 object o_3 is expected to be inside the query result of query q_1. Thus object o_3 should not be included in the predictive result 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 predictive result at time t_1. Since o_2 has not changed its MSB, it remains included in query q_1's predictive result. By applying the valid prediction time interval test on o_4, we obtain a non-empty time interval with respect to query q_1, during which o_4 is expected to be included in the query result. Thus o_4 is added into the predictive result of query q_1.

3.4 Setting α and β Values

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* (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.

In order to choose appropriate α and β values for each MSB, 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 settings is to develop an analytical model and use the analytical results to guide the runtime selection of the best parameter settings. We develop an analytical model for estimating the IO cost of performing query evaluation (see Appendix for details). This model is used as the guide to build an off-line 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. We will discuss details of α and β value selection in Section 3.6.

3.5 Motion Adaptive Indexing

We have described the main ideas and mechanisms used in our motion-adaptive indexing scheme. 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.

3.5.1 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 maintain the query results as objects and the spatial regions of the queries move. The detailed description of the two tables is given below:

Moving Object Table (MOT): An MOT entry is described as $(oid, qid, pos, vel, time, box, P_{cm}, V_{ch})$ and stores information regarding a moving object. Here, oid is the moving object identifier, qid is the query identifier of the moving query whose focal object's identifier is oid, qid is $null$ if no such moving query exists, pos is the last received position, vel is the last received velocity vector of the moving object, $time$ is the timestamp of the motion updates (pos and vel) received from the moving object, box is the MSB of the moving object, P_{cm} is an estimate on the

Algorithm 1: Motion Update Received $r = \langle oid, pos, vel, time \rangle$
$eo = \langle oid, qid, pos, vel, time, box, P_{cm}, V_{ch} \rangle \in MOT$
where $eo.oid = r.oid$
$eo.P_{cm} \leftarrow \gamma * (r.time - eo.time) + (1 - \gamma) * eo.P_{cm}$
$eo.pos \leftarrow r.pos$
$eo.vel \leftarrow r.vel$
$eo.time \leftarrow r.time$
$eo.V_{ch} \leftarrow$ **true**
if $eo.qid \neq$ **null then**
$eq = \langle qid, pos, vel, radius, time, box, P_{cm}, V_{ch} \rangle \in MQT$
where $eq.qid = eo.qid$
$eq.P_{cm} = \gamma * (r.time - eq.time) + (1 - \gamma) * eq.P_{cm}$
$eq.pos \leftarrow r.pos$
$eq.vel \leftarrow r.vel$
$eq.time \leftarrow r.time$
$eq.V_{ch} \leftarrow$ **true**
end if

period of constant motion of the object and V_{ch} is a Boolean variable indicating whether the moving object has changed its motion function since the last query evaluation step.

Moving Query Table (MQT): An MQT entry is described as $(qid, pos, vel, radius, time, box, P_{cm}, V_{ch})$ and stores information regarding a moving query. Here, qid is the moving query identifier, pos and vel are the last received position and the last received velocity vector of the query's focal object respectively, $time$ is the timestamp of the motion updates (pos and vel) received from the focal object, $radius$ is the radius of the moving query's spatial region, box is the MSB of the moving query, P_{cm} is an estimate on the period of constant motion 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 P_{cm} entries are updated using a simple weighted running average. The details are given in Algorithm 1. 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.

3.5.2 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 differentially 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 or changed its motion function since the last query evaluation period. 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 (See Section 3.6 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. Algorithm 2 gives the pseudo code for the MOT scan.

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 or its focal object has changed its motion function since the last query evaluation step. 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 (See Section 3.6 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. Algorithm 3 gives the pseudo code for the MQT scan.

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. The order of the scans can be reversed with some minor modifications.

3.6 $\alpha\beta Table$ and Adaptive Parameter Selection

The cost function developed in Appendix 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 and period of constant motion, 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 3.5, 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

Algorithm 2: Moving Object Table Scan

for all $e = \langle oid, qid, pos, vel, time, box, P_{cm}, V_{ch} \rangle \in MOT$ **do**
  $ctime \leftarrow$ current time
  $e.pos \leftarrow e.pos + (ctime - e.time) * e.vel$
  $e.time \leftarrow ctime$
  $boxInvalid \leftarrow cpos \notin e.box$
  if $\neg boxInvalid \wedge \neg e.V_{ch}$ **then**
    continue
  end if
  $oldBox \leftarrow e.box$
  if $e.V_{ch}$ **then**
    $e.V_{ch} \leftarrow$ **false**
    if $\neg boxInvalid$ **then**
      {Update the data associated with $e.oid$ in the $Index_o^{msb}$}
      $oidx.updateData(oid, \langle e.pos, e.vel, e.time \rangle)$
    end if
  end if
  if $boxInvalid$ **then**
    $\alpha \leftarrow \alpha\beta Table.lookup(|e.vel|, e.P_{cm})$
    $e.box \leftarrow Rect(e.pos, e.pos + \alpha * e.vel)$
    {Update the entry associated with $e.oid$ in the $Index_o^{msb}$}
    $oidx.update(oid, e.box, \langle e.pos, e.vel, e.time \rangle)$
  end if
  $oldQids \leftarrow qidx.query(oldBox)$
  $newResults \leftarrow qidx.query(e.box, e.pos, e.vel, e.time)$
  for all $r = \langle qid, itv = \langle itvs, itve \rangle \rangle \in newResults$ **do**
    $result \leftarrow results.get(r.qid)$
    $result.put(e.oid, itv)$
    $oldQids.remove(r.qid)$
  end for
  for all $qid \in oldQids$ **do**
    $result \leftarrow results.get(r.qid)$
    $result.remove(e.oid)$
  end for
end for

Algorithm 3: Moving Query Table Scan

for all $e = \langle qid, pos, vel, radius, time, box, P_{cm}, V_{ch} \rangle \in MQT$ **do**
  $ctime \leftarrow$ current time
  $e.pos \leftarrow e.pos + (ctime - e.time) * e.vel$
  $e.time \leftarrow ctime$
  $boxInvalid \leftarrow cpos \notin e.box$
  if $\neg boxInvalid \wedge \neg e.V_{ch}$ **then**
    continue
  end if
  if $e.V_{ch}$ **then**
    $e.V_{ch} \leftarrow$ **false**
    if $\neg boxInvalid$ **then**
      {Update the data associated with $e.qid$ in the $Index_q^{msb}$}
      $qidx.updateData(qid, \langle e.pos, e.vel, e.radius, e.time \rangle)$
    end if
  end if
  if $boxInvalid$ **then**
    $\beta \leftarrow \alpha\beta Table.lookup(|e.vel|, e.P_{cm})$
    $fpos \leftarrow e.pos + \beta * e.vel$
    $e.box \leftarrow Rect(e.pos - sign(e.vel) * e.radius, fpos + sign(e.vel) * e.radius)$
    {Update the entry associated with $e.qid$ in the $Index_q^{msb}$}
    $qidx.update(qid, e.box, \langle e.pos, e.vel, e.radius, e.time \rangle)$
  end if
  $result \leftarrow results.get(e.qid)$
  $result.removeAll()$
  $newResults \leftarrow oidx.query(e.box, e.pos, e.vel, e.radius, e.time)$
  for all $r = \langle oid, itv = \langle itvs, itve \rangle \rangle \in newResults$ **do**
    $result.put(r.oid, itv)$
  end for
end for

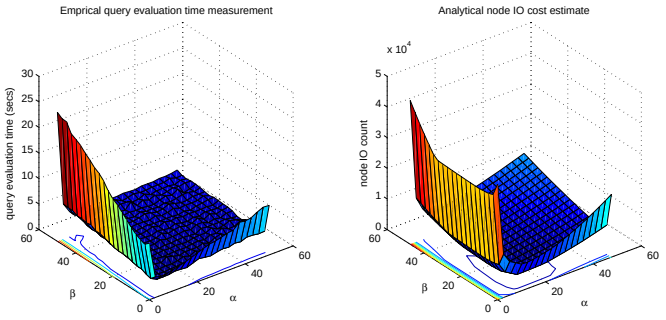

Figure 4: Analytical node IO estimate and experimental query evaluation time

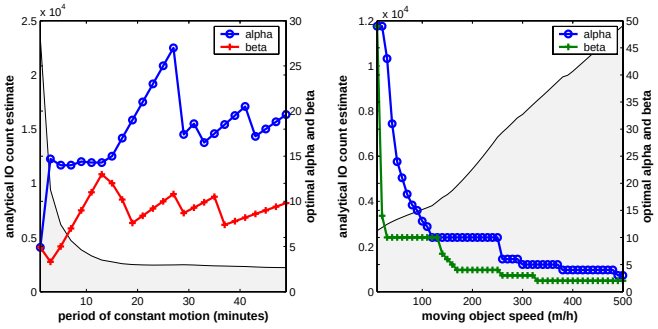

Figure 5: Optimal α and β values

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 off-line computed $\alpha\beta Table$.

The graph on the left in Figure 4 plots the average time it takes to perform one complete query evaluation step (labelled as *total query evaluation time*) as a function of α and β. These values are from the actual implementation of motion adaptive indexing. The graph on the right in Figure 4 plots the analytical node IO count estimate of performing one query evaluation step as a function of α and β. Two important observations can be obtained by comparing these graphs. First, it shows 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 results in faster query evaluation.

The graph on the left in Figure 5 plots the optimal α and β values calculated by the analytical cost estimate (using the right y-axis), as a function of period of constant motion. The node IO count is also shown in the graph as an area chart (using the left y-axis). Note that as the period of constant motion increases, the object movements are more predictable. For small values of the period of constant motion, the optimal β value turns out to be small. This is because large β values will result in more prediction, which is not desirable when the period of constant motion is small (predictability is poor).

The graph on the right in Figure 5 plots the optimal α and β values calculated by the analytical cost estimate (using the right y-axis), as a function of object speeds. The node IO is also shown in the graph as an area chart (using the left y-axis). The graph shows that 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.

More experiments on the effect of adaptive parameter selection will be provided in Section 5.

4 Evaluating Moving kNN Queries

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.

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 radiuses, we need to make the following three changes:

a. 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.

b. 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 MQ with its range equal to the safe radius.

c. At the end of each query evaluation step, 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 distance between the object and the focal object of the query *at the end* of the 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 too large.

The optimistic safe radius approach retrieves the current k nearest neighbors, and for each object in the list calculates the maximum distance between the object and the focal object of the query *throughout* the next t time units. 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.

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 over fixed parameter setting. 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.

Parameter	Default value
area of the region of interest	500000 square miles
number of objects	50000
percentage of moving objects	50%
number of queries	5000
percentage of moving queries	50%
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 10 minutes geometrically distributed
moving object speed	between 0-160 miles/hour uniformly random
scan period	30 seconds
motion update time period	30 seconds

Table 1: System Parameters

5.1 System Parameters and Setup

We list the set of parameters used in the experiment in Table 1. In all of the experiments presented in the rest of the paper, the parameters take their default values if not specified otherwise. The default object density is taken in accordance with previous work [18, 19]. 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.

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. 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.

5.2 Performance Comparison

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* MSB*s* (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. Object Indexing with MSBs is similar to pure object-only indexing, except that the motion sensitive boxes are used in the place of objects in the spatial index (without the predictive query results).

Figure 6 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 7 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

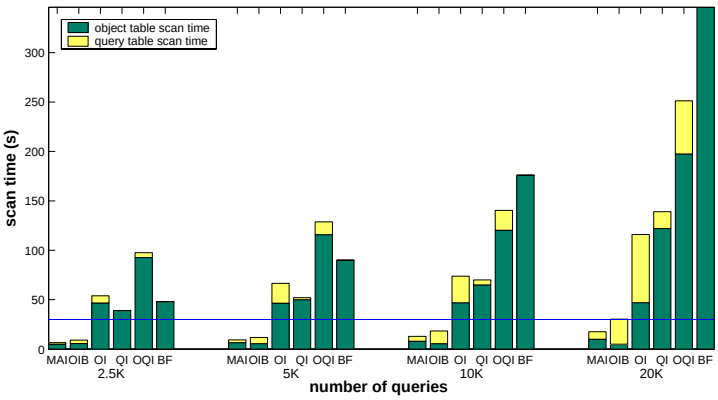

Figure 6: Query evaluation time

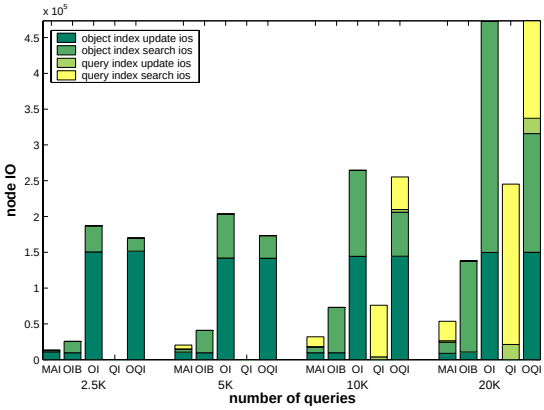

Figure 7: Query evaluation node IO

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 7. Several observations can be obtained from Figure 6 and Figure 7.

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 6. 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 7.

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 6. 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 7.

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 6. 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 1 hour) and move fast (max. speed 500 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 8 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 8.

First, we notice that the adaptive parameter selection has a clear performance advantage. This is clearly observed from Figure 8, 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.

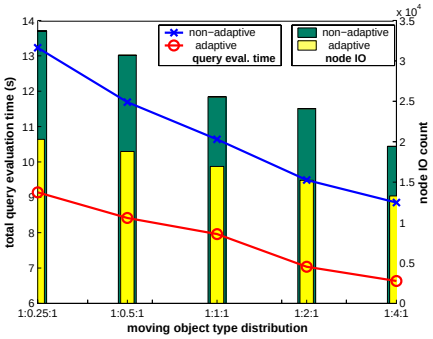

Figure 8: Performance gain due to adaptive parameter selection

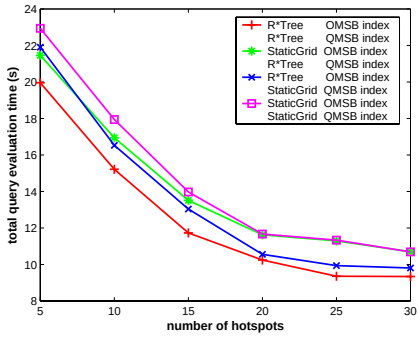

Figure 9: Effect of data and query skewness on performance

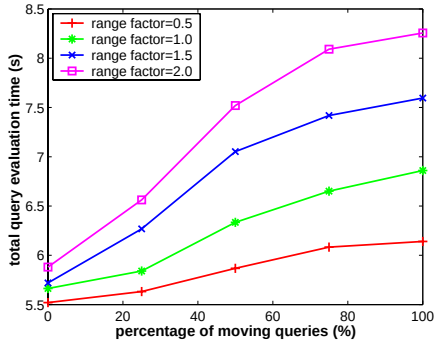

Figure 10: Effect of query range & moving query % on performance

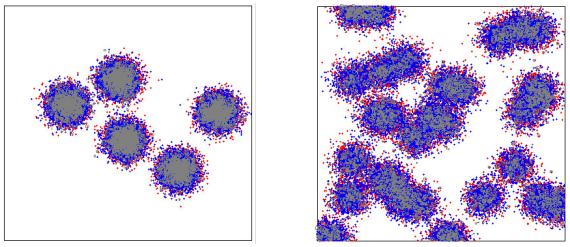

Figure 11: Query and object distribution for $N_h = 5$ and $N_h = 30$

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. Figure 11 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 [7]. 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 [9] 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 MQs.

Figure 9 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 9 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 9 also shows that R*-tree performs the best under all conditions. But during our experiments we also observed that without using the optimization discussed under R*-tree implementation detail (in the Appendix) for decreasing the update operation cost of the R*-tree, the results are in favor of the static grid. Put differently, our experiments conclude that only a properly implemented R*-tree with optimized update operation outperforms a static grid based approach.

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 10 plots the total query evaluation scan time as a function of moving query percentage for different range factors. As shown in Figure 10, 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. On the other hand, increasing the range factor shows roughly linear increase on the query evaluation time.

In Figure 12 we study the effect of the number of objects on the query evaluation performance. Figure 12 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 12 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.

5.6 Performance of Continual kNN Queries

We compare the performance of MQ based moving continual kNN query evaluation against the object-only indexing approach. In object-only indexing approach, the object index is updated and the kNN queries are evaluated against the updated object index during each query evaluation step. In this experiment 10K objects are used with the same object density (N_o/A) specified in Sec-

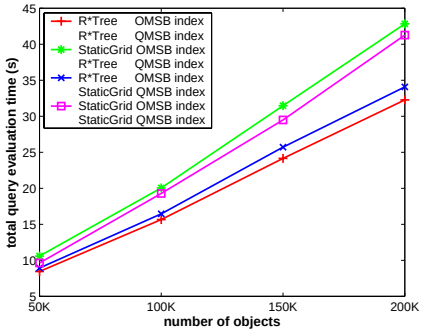

Figure 12: Effect of number of objects on performance

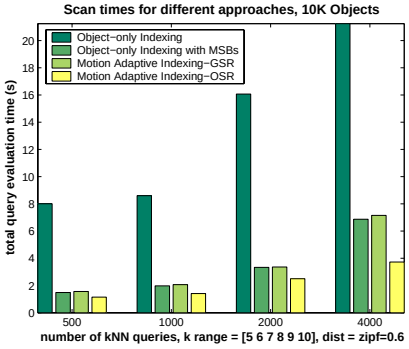

Figure 13: Total query evaluation time for moving continual kNN queries

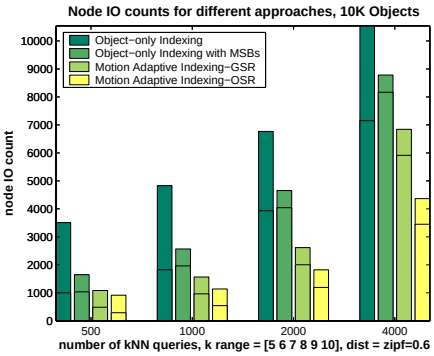

Figure 14: Node IO count for moving continual kNN queries

tion 5.1), where 50% of the objects are moving with the default motion parameters from Section 5.1. All queries are moving continual kNN queries and the number of queries ranges from 0.5K to 4K. 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. Figure 13 plots the total query evaluation time and Figure 14 plots the node IO count for different number 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 object-only indexing approach. Between the two variations of safe radius, OSR (optimistic safe radius based approach) performs better than GSR (guaranteed safe radius based approach). Object-only indexing with MSBs (OIB) slightly outperforms GSR. However, OSR provides 20-40% improvement in total query evaluation time over OIB.

6 Related Work

Research on moving object indexing can be broadly divided into two categories, (1) indexing and querying current positions of moving objects and (2) indexing and querying trajectories of 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 [17]. Continual queries are used as a useful tool for monitoring frequently changing information [23, 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 [18, 9, 11, 3, 19] or on moving continual queries over static objects [21, 2, 20].

In [18], velocity constrained indexing and query indexing has been proposed for efficient evaluation of static continual range queries. The same problem is studied in [9], however the focus is on in-memory structures and algorithms. In [19], TPR-tree, an R-tree based indexing structure, is proposed for efficient evaluation of spatial queries over moving object positions. TPR* tree, an extension of TPR tree optimized for queries that look into future (predictive), is described in [22]. Work on moving continual queries over static objects focuses on continuous k-nearest neighbor evaluation. An algorithm for precalculating k-nearest neighbors with a line segment representing the continuous motion of an object, is described in [21]. In [2], reverse nearest neighbors are also discussed.

The concept of moving queries is to some extent similar to the Dynamic Queries introduced in [12]. A dynamic query is defined as a temporally ordered set of snapshot queries in [12]. This is a low level definition as opposed to our definition of moving queries which is more declarative and is defined from users' perspective. The work done in [12] 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 moving objects involved.

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 promote to index less frequently changing MSBs 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 to use *motion adaptive* indexing in the sense that the sizes of the MSBs 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.

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] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *ACM PODS*, 1994.

[5] C. Faloutsos, T. K. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *ACM SIGMOD*, 1987.

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

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

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

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

[10] I. Kamel and C. Faloutsos. On packing R-trees. In *ACM CIKM*, 1993.

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

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

[13] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-trees: A bottom-up approach. In *VLDB*, 2003.

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

[15] B. Pagel, H. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *ACM PODS*, 1993.

[16] 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.

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

[18] 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.

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

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

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

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

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

[24] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *ACM PODS*, 1996.

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

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

[27] 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.

P_s	scan period	R_{mq}	average moving query radius
P_{cm}	period of constant motion	L_{sq}	average static query side length
N_o	number of objects	S_{mo}	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

Table 2: Symbols and their meanings

APPENDIX - Analytical Model for IO Estimation and $\alpha\beta$ Setting

In this section we develop an analytical model for estimating the IO cost of performing query evaluation and describe how we use this model to set α and β parameters adaptively based on the dynamic motion behavior of the moving objects or moving queries. Although our approach is spatial index independent, the discussion on the analytical model is based on R*-trees.

R*-tree Implementation Details

We have used an R*-tree implementation which can store both spatial objects and their associated data (motion functions in our case) at the leaf level. The only modification we have performed is on the update operation. For moving object indexing, performing an update as a delete operation followed by an insert operation is costly. Most of the time the position change of the spatial index entry do not cause any inconsistencies at the leaf level, i.e. the MBR of the entry's leaf node will still contain its new position. As a result our update operation implementation first checks whether the position change of the spatial index entry causes its new MBR to cross its current leaf node's MBR. If not, the position of the entry is updated without causing any structural change on the tree. Otherwise a delete operation followed by an insert operation is performed as the fallback plan. In fact, the update operation can be further improved by using a bottom up approach as recently introduced in [13].

Analytical Model for IO Estimation

In what follows, we present an analytical model to calculate the IO

cost of the query evaluation, i.e. the two scans performed at each query evaluation step. 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,

$$\begin{aligned} A_{mo} &\approx (\alpha * S_{mo}/\sqrt{2})^2 \\ A_{mq} &\approx (\beta * S_{mo}/\sqrt{2} + 2 * R_{mq})^2 \end{aligned}$$

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,

$$\begin{aligned} A_o &\approx A_{mo} * N_{mo}/N_o \\ A_q &\approx (A_{mq} * N_{mq} + L_{sq}^2 * (N_q - N_{mq}))/N_q \end{aligned}$$

Given this information, the following four quantities can be analytically derived based on well studied R-tree cost models [5, 10, 15, 4, 24, 25]:

C_o^u: node IO cost for updating the $Index_o^{msb}$ during the processing of an object table entry

C_o^s: node IO cost for searching the $Index_q^{msb}$ during the processing of an object table entry

C_q^u: node IO cost for updating the $Index_q^{msb}$ during the processing of a query table entry

C_q^s: node IO cost for searching the $Index_o^{msb}$ during the processing of a query table entry

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,

$$\begin{aligned} N_o^{vc} &\approx N_{mo} * \min(1, P_s/P_{cm}) \\ N_q^{vc} &\approx N_o^{vc} * N_{mq}/N_{mo} \end{aligned}$$

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,

$$\begin{aligned} N_o^{bi} &\approx \min(P_s/\alpha, 1) * N_{mo} \\ N_q^{bi} &\approx \min(P_s/\beta, 1) * N_{mq} \end{aligned}$$

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 (VCE for short) 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,:

$$\begin{aligned} N_{mot} &\approx N_o^{vc} + N_o^{bi} - N_o^{bi} * Prob\{\text{object caused VCE}\} \\ N_{mqt} &\approx N_q^{vc} + N_q^{bi} - N_q^{bi} * Prob\{\text{query caused VCE}\} \end{aligned}$$

Then,

$$\begin{aligned} N_{mot} &\approx N_o^{vc} + N_o^{bi} - N_o^{bi} * (N_o^{vc}/N_{mo}) \\ N_{mqt} &\approx N_q^{vc} + N_q^{bi} - N_q^{bi} * (N_q^{bi}/N_{mq}) \end{aligned}$$

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:

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

Setting values of α and β

The cost function given by (1) has a global minimum with respect to parameters α and β for fixed values of moving object speed and constant motion time. This fact is used to construct the $\alpha\beta$Table mentioned in Section 3.6.