Appeared in *IEEE Transactions on Computers*, Vol. 51, No. 10, pp. 1124-1140, Oct 2002

Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects *

S. Prabhakar Y. Xia D. Kalashnikov W. G. Aref S. Hambrusch
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907
U.S.A.
E-mail: {sunil,xia,dvk,aref,seh}@cs.purdue.edu

Keywords: *Moving Objects, Spatio-Temporal Indexing, Continuous Queries, Query Indexing*.

Abstract

Moving object environments are characterized by large numbers of moving objects and numerous concurrent continuous queries over these objects. Efficient evaluation of these queries in response to the movement of the objects is critical for supporting acceptable response times. In such environments the traditional approach of building an index on the objects (data) suffers from the need for frequent updates and thereby results in poor performance. In fact, a brute force, no-index strategy yields better performance in many cases. Neither the traditional approach, nor the brute force strategy achieve reasonable query processing times. This paper develops novel techniques for the efficient and scalable evaluation of multiple continuous queries on moving objects. Our solution leverages two complimentary techniques: *Query Indexing* and *Velocity Constrained Indexing* (*VCI*). Query Indexing relies on i) incremental evaluation; ii) reversing the role of queries and data; and iii) exploiting the relative locations of objects and queries. VCI takes advantage of the maximum possible speed of objects in order to delay the expensive operation of updating an index to reflect the movement of objects. In contrast to an earlier technique [29] that requires exact knowledge about the movement of the objects, VCI does not rely on such information. While Query Indexing outperforms VCI, it does not efficiently handle the arrival of new queries. Velocity constrained indexing, on the other hand, is unaffected by changes in queries. We demonstrate that a combination of Query Indexing and Velocity Constrained Indexing enables the scalable execution of insertion and deletion of queries in addition to processing ongoing queries. We also develop several optimizations and present a detailed experimental evaluation of our techniques. The experimental results show that the proposed schemes outperform the traditional approaches by almost two orders of magnitude.

1 Introduction

The combination of personal locator technologies [19, 34], global positioning systems [23, 33], and wireless [11] and cellular telephone technologies enables new location-aware services, including location and mobile commerce (L- and M-commerce). Current location-aware services allow proximity-based queries including map viewing and navigation, driving directions, searches for hotels and restaurants, and weather and traffic information. They include GPS based systems like Vindigo and SnapTrack and cell-phone based systems like TruePosition and Cell-Loc.

These technologies are the foundation for pervasive location-aware environments and services. Such services have the potential to improve the quality of life by adding location-awareness to virtually all objects of interest such as humans, cars,

* Work Supported by NSF CAREER Grant IIS-9985019, NSF Grants 9988339-CCR, 9972883, and 0010044-CCR, a Gift from Microsoft Corp.

laptops, eyeglasses, canes, desktops, pets, wild animals, bicycles, and buildings. Applications can range from proximity-based queries on non-mobile objects, locating lost or stolen objects, tracing small children, helping the visually challenged to navigate, locate, and identify objects around them, and to automatically annotating objects online in a video or a camera shot. Examples of such services are emerging for locating persons [19] and managing emergency vehicles [21]. These services correspond to queries that are executed over an extended period of time (i.e. from the time they are initiated to the time at which the services are terminated). During this time period the queries are repeated evaluated in order to provide the correct answers as the locations of objects change. We term these queries *Continuous Queries*. A fundamental type of continuous query required to support many of the services mentioned above is the range query.

Our work assumes that objects report their current location to stationary servers. By communicating with these servers, objects can share data with each other and discover information (including location) about specified and surrounding objects. Throughout the paper, the term "object" refers to an object that (a) knows its own location and (b) can determine the locations of other objects in the environment through the servers.

This paper develops novel techniques for the efficient and scalable evaluation of multiple continuous range queries on moving objects. Our solution leverages two complimentary techniques: *Query Indexing* and *Velocity Constrained Indexing*. Query Indexing gives almost two orders of magnitude improvement over traditional techniques. It relies on i) incremental evaluation; ii) reversing the role of queries and data; and iii) exploiting the relative locations of objects and queries. Velocity constrained indexing (VCI) enables efficient handling of changes to queries. VCI allows an index to be useful even when it does not accurately reflect the locations of objects that are indexed. It relies upon the notion of maximum speeds of objects. Our model of object movement makes no assumptions for query-indexing. For the case of VCI, we assume only that each object has a maximum velocity that it will not exceed. If necessary, this value can be changed over time. We do not assume that objects need to report and maintain a fixed speed and direction for any period of time as in [29]. The velocity constrained index remains effective for large periods of time without the need for any updates, independent of the actual movement of objects. Naturally, its effectiveness drops over time and infrequent updates are necessary to counter this degradation. A combined approach of these two techniques enables the scalable execution of insertion and deletion of queries in addition to processing ongoing queries. We also develop several optimizations for: (i) reducing communication and evaluation costs for Query Indexing – *safeRegions*; (ii) efficient post-processing with VCI through *Clustering*; and (iii) efficient updates to VCI – *Refresh* and *Rebuild*. A detailed experimental evaluation of our techniques is conducted. The experimental results demonstrate the superior performance of our indexing methods as well as their robustness to variations in the model parameters.

Our work distinguishes itself from related work in that it addresses the issues of scalable execution of concurrent continuous queries (as the numbers of mobile objects and queries grow). This paper argues that the traditional query processing approaches where objects are indexed and queries are posed to these indexes may not be the relevant paradigm in moving object environments. Due to the large numbers of objects that move, the maintenance of indexes tends to be very expensive. In fact, as our experiments demonstrate, these high costs make the indexes more inefficient than simple scans over the entire data, even for 2-dimensional data.

The rest of this paper proceeds as follows. Related work is discussed in Section 2. Section 3 describes the traditional solution and our assumptions about the environment. Section 4 presents the approach of Query Indexing and related optimizations. The alternative scheme of Velocity Constrained Indexing is discussed in Section 5. Experimental evaluation of the proposed schemes is presented in Section 6, and Section 7 concludes the paper.

2 Related Work

The growing importance of moving object environments is reflected in the recent body of work addressing issues such as indexing, uncertainty management, broadcasting, and models for spatio-temporal data. To the best of our knowledge no existing work addresses the timely execution of multiple concurrent queries on a collection of moving objects as proposed

in the following sections. We do not make any assumption about the future positions of objects. It is also not necessary for objects to move according to well behaved patterns as in [29]. In particular, the only constraint imposed on objects in our model is that for Velocity Constrained Indexing (discussed in Section 5) each object has a maximum speed at which it can travel (in any direction).

Indexing techniques for moving objects are being proposed in the literature, e.g., [8, 20] index the histories, or trajectories, of the positions of moving objects, while [29] indexes the current and anticipated future positions of the moving objects. In [18], trajectories are mapped to points in a higher-dimensional space which are then indexed. In [29], objects are indexed in their native environment with the index structure being parameterized with velocity vectors so that the index can be viewed at future times. This is achieved by assuming that an object will remain at the same speed and in the same direction until an update is received from the object.

Uncertainty in the positions of the objects is dealt with by controlling the update frequency [24, 37], where objects report their positions and velocity vectors when their actual positions deviate from what they have previously reported by some threshold. Tayeb et. al. [32] use quadtrees [30] to index the trajectories of one-dimensional moving points. Kollios [18] et. al. map moving objects and their velocities into points and store the points in a kD-tree. Pfoser et. al. [26, 25] index the past trajectories of moving objects that are presented as connected line segments. The problem of answering a range query for a collection of moving objects is addressed in [3] through the use of indexing schemes using external range trees. [36, 38] consider the management of collections of moving points in the plane by describing the current and expected positions of each point in the future. They address how often to update the locations of the points to balance the costs of updates against imprecision in the point positions. Spatio-temporal database models to support moving objects, spatio-temporal types and supporting operations have been developed in [12, 13].

Scalable communication in the mobile environment is an important issue. This includes location updates from objects to the server and relevant data from the server to the objects. Communication is not the focus of this paper. We propose the use of *Safe Regions* to minimize communication for location updates from objects. We assume that the process of dissemination of safe regions is carried out by a separate process. In particular, this can be achieved by a periodic broadcast of safe regions. Efficient broadcast techniques are proposed in [1, 2, 14, 15, 16, 17, 40]. In particular, the issue of efficient (in terms of battery-time and latency) broadcast of indexed multi-dimensional data (such as safe regions) is addressed in [14].

3 Moving Object Environment

3.1 Pervasive Location-Aware Computing Environments

Figure 1 sketches a possible hierarchical architecture of a location-aware computing environment. Location detection devices (e.g., GPS devices) provide the objects with their geographical locations. Objects connect directly to regional servers. Regional servers can communicate with each other, as well as with the repository servers. Data regarding past locations of objects can be archived at the repository servers. We assume that (i) the regional servers and objects have low bandwidth and a high cost per connection, and (ii) repository servers are interconnected by high bandwidth links. This architecture is similar to that of current cellular phone architectures [31, 35]. For information sent to the objects, we consider point-to-point communication as well as broadcasting. Broadcasting allows a server to send data to a large number of “listening” objects [1, 2, 14, 15, 16, 40]. Key factors in the design of the system are scalability with respect to large numbers of objects and the efficient execution of queries.

In traditional applications, GPS devices tend to be passive i.e., they do not exchange any information with other devices or systems. More recently, GPS devices are becoming active entities that transmit and receive information that is used to affect processing. Examples of these new applications include vehicle tracking [21], identification of closest emergency vehicles in Chicago [21], and Personal Locator Services [19]. Each of these examples represents commercial developments that handle small scale applications. Another example of the importance of location information is the emerging Enhanced

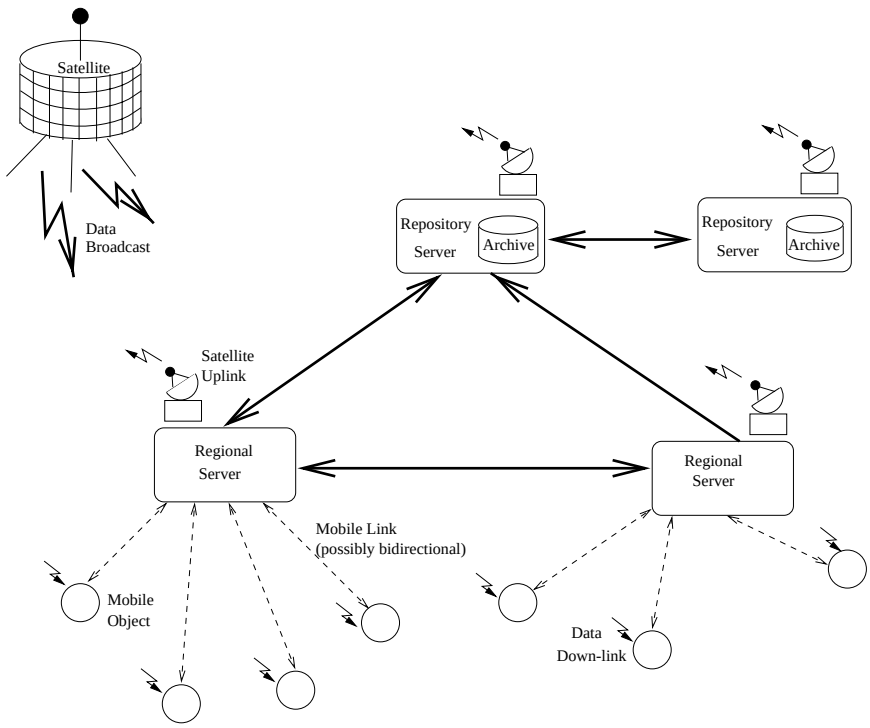


Figure 1: Illustrating a location-aware environment

911 (E911) [39] standard. The standard seeks to provide wireless users the same level of emergency 911 support as wireline callers. It relies on wireless service providers calculating the approximate location of the cellular phone user. The availability of location-awareness would further enhance the ability of emergency services to respond to a call e.g., using medical history of the caller. Applications such as these, improvements in GPS technology, and reducing cost, augur the advent of pervasive location-aware environments. The PLACE (Pervasive Location-Aware Computing Environments) project at Purdue University is addressing the underlying issues of query processing and data management for the moving object environments [28]. Connectivity is achieved through wireless links as well as mobile telephone services.

3.2 Continuous Query Processing

Location-aware environments are characterized by large numbers of moving (and stationary) objects. These environments will be expected to provide several types of location centric services to users. Examples of these services include: navigational services that aid the user in understanding her environment as she travels; subscription services wherein a user identifies objects or regions of interest and is continuously updated with information about them; and group management services that enable the coordination and tracking of collections of objects or users. To support these services it is necessary to execute efficiently several types of queries, including range queries, nearest-neighbor queries, density queries, etc. An important requirement in location-aware environments is the continuous evaluation of queries. Given the large numbers of queries and moving objects in such enviroments, and the need for a timely response for continuous queries, efficient and scalable query execution is paramount.

In this paper we focus on range queries. The solutions need to be scalable in terms of the number of total objects, degree of movement of objects, and the number of concurrent queries. Range queries arise naturally and frequently in spatial applications such as a query that needs to keep track of, for example, the number of people that have entered a building. Range queries can also be useful as pre-processing tools for reducing the amount of data that other queries, such as nearest-

neighbor or density, need to process.

3.3 Model

In our model, objects are represented as points, and queries are expressed as rectangular spatial regions. Therefore, given a collection of moving objects and a set of queries, the problem is to identify which objects lie within (i.e., are relevant to) which queries. We assume that objects report their new locations to the server periodically or when they have moved by a significant distance. Updates from different objects arrive continuously and asynchronously at the server. The location of each object is saved in a file on the server. Since all schemes incur the cost of updating this file and the updating is done in between the evaluation intervals, we do not consider the cost of updating this file as objects move. Objects are required to report only their location, not the velocity. There is no constraint on the movement of objects except that the maximum possible speed of each object is known and not exceeded (this is required only for Velocity Constrained Indexing). We expect that at any given time only a small fraction of the objects will move.

Ideally, each query should be re-evaluated as soon as an object moves. However, this is impractical and may not even be necessary from the user's point of view. We therefore assume that the continuous evaluation of queries takes place in a periodic fashion whereby we determine the set of objects that are relevant to each continuous query at fixed time intervals. This interval, or time step, is expected to be quite small (e.g. in [18] it is taken to be 1 minute) – our experiments are conducted with a time interval of 50 seconds.

3.4 Limitations of Traditional Indexing

In this section we discuss the traditional approaches to answering queries for moving objects and their limitations. Our approaches are presented in Sections 4 and 5.

A *brute force* method to determine the answer to each query compares each query with each object. This approach does not make use of the spatial location of the objects or the queries. It is not likely to be a scalable solution given the large numbers of moving objects and queries.

Since we are testing for spatial relationships, a natural alternative is to build a spatial index on the objects. To determine which objects intersect each query, we execute the queries on this index. All objects that intersect with a query are relevant to the query. The use of the spatial index should avoid many unnecessary comparisons of queries against objects and thereby we expect this approach to outperform the brute force approach. This is in agreement with conventional wisdom on indexing. In order to evaluate the answers correctly, it is necessary to keep the index updated with the latest positions of objects as they move. This represents a significant problem. Notice that for the purpose of evaluating continuous queries, we are not interested in preserving the historical data but rather only in maintaining the current snapshot. The historical record of movement is maintained elsewhere such as at a repository server (see Figure 1).

In Section 6 we evaluate three alternatives for keeping the index updated. As we will see in Section 6, each of these gives very poor performance. The poor performance of the traditional approach of building an index on the data (i.e. the objects) can be traced to the following two problems: i) whenever *any* object moves, it becomes necessary to re-execute *all* queries; and ii) the cost of keeping the index updated is very high. In the next two sections we develop two novel indexing schemes that overcome these limitations.

4 Query Indexing: Queries as Data

The traditional approach of using an index on object locations to efficiently process queries for moving objects suffers from the need for constant updates to the index and re-evaluation of all queries whenever any object moves. We propose an alternative that addresses these problems based upon two key ideas:

- treating *queries as data* and the data as queries, and
- *incremental* evaluation of continuous queries.

We also develop the notion of *safe regions* that exploit the relative location of objects and queries to further improve performance.

In treating the queries as data, we build a spatial index such as an R-tree on the queries instead of the customary index that is built on the objects (i.e. data). We call this the Query-Index or *Q-index*. To evaluate the intersection of objects and queries, we treat each object as a "query" on the Q-index (i.e., we treat the moving objects as queries in the traditional sense). Exchanging queries for data results in a situation where we execute a larger number of queries (one for each object) on a smaller index (the Q-index), as compared to an index on the objects. This is not necessarily advantageous by itself. However, since not all objects change their location at each time step, we can avoid a large number of "queries" on the Q-index by incrementally maintaining the result of the intersection of objects and queries.

Incremental evaluation is achieved as follows: upon creation of the Q-index, all objects are processed on the Q-index to determine the initial result. Following this, we incrementally adjust the query results by considering the movement of objects. At each evaluation time step, we process only those objects that have moved since the last time step, and adjust their relevance to queries accordingly. If most objects do not move during each time step, this can greatly reduce the number of times the Q-index is accessed. For objects that move, the Q-index improves the search performance as compared to a comparison against all queries.

Under the traditional indexing approach, at each time step, we would first need to update the index on the objects (using one of the alternatives discussed above) and then evaluate each query on the modified index. This is independent of the movement of objects. With the "Queries as Data" or the Q-index approach, only the objects that have moved since the previous time step are evaluated against the Q-index. Building an index on the queries avoids the high cost of keeping an object index updated; incremental evaluation exploits the smaller numbers of objects that move in a single time step to avoid repeating unnecessary comparisons. Upon the arrival of a new query, it is necessary to compare the query with all the objects in order to initiate the incremental processing. Deletion of queries is easily handled by ignoring those queries.

Further improvements in performance can be achieved by taking into account the relative locations of objects and queries. Next we present optimizations based upon this approach.

4.1 *Safe Regions*: Exploiting Query and Object Locations

Consider an object that is far away from any query. This object has to move a large distance before its relevance to any query changes. Let $SafeDist$ be the shortest distance between object O and a query boundary. Clearly, O has to move a distance of at least $SafeDist$ before its relevance with respect to any query changes. Thus we need not check the Q-index with O's new location as long as it has not moved by $SafeDist$. Similarly, we can define two other measures of "safe" movement for each object:

- *SafeSphere* – a safe sphere (circle for two dimensions) around the current location. The radius of this sphere is equal to the *SafeDist* discussed above.
- *SafeRect* – a safe maximal rectangle around the current location. Maximality can be defined in terms of rectangle area, perimeter, etc.

Figure 2 shows examples of each type of *Safe Region*. Note that it is not important whether an object lies within or outside a query that contributes to its safe region. Points X and Y are examples of each type of point: X is not contained within any query, whereas Y is contained in query Q_1. The two circles centered at X and Y are the *SafeSphere* regions for X and Y respectively, and the radii of the two circles are their corresponding *SafeDist* values. Two examples of *SafeRect* are shown for X. The *SafeRect* for Y is within Q_4. Note that for X, other possibilities for *SafeRect* are possible. With each

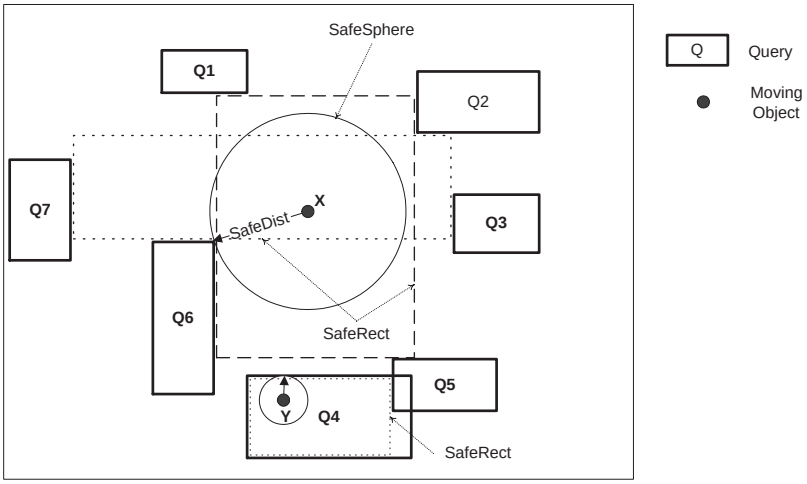


Figure 2: Examples of *Safe Regions*

approach, only objects that move out of their safe region need to be evaluated against the Q-index. These measures identify ranges of movement for which an object's matching does not change and thus it need not be checked against the Q-index. This significantly reduces the number of accesses to Q-index. Note that for the *SafeDist* technique, we need to keep track of the total distance traveled since *SafeDist* was computed. Once an object has traveled more than *SafeDist*, it needs to be evaluated against the Q-index until *SafeDist* is recomputed. On the other hand, for the *SafeSphere* and *SafeRect* measures, an object could exit the safe region, and then re-enter it at a later time. While the object is inside the safe region it need not be evaluated against Q-index. While it is outside the safe region, it must be evaluated at each time step.

The safe region optimizations significantly reduce the need to test data points for relevance to queries if they are far from any query boundaries and move slowly. Recall that each object reports its location periodically or when it has moved by a significant distance since its last update. This decision can be based upon safe region information sent to each object. Thus the object need not report its position when it is within the safe region, thereby reducing communication and the need for processing at the server. The effectiveness of these techniques in reducing the number of objects that need to report their movement is studied in Section 6. Even though we do not perform any re-computation of the safe regions in our experiments, we find that the safe region optimizations are very effective. It should be noted that multiple safe regions can be combined to produce even larger safe regions. By definition, there are no query boundaries in a safe region. Hence there can be no query boundary in the union of the two safe regions.

4.2 Computing the Safe Regions

The Q-index can be used to efficiently compute each of the safe regions. *SafeDist* is closely related to a nearest-neighbor query since it is the distance to the nearest query boundary. A branch-and-bound algorithm similar to that proposed for nearest neighbor queries in [27] is used. The [27] algorithm prunes the search based upon the distances to queries and bounding boxes that have already been visited. Our *SafeDist* algorithm is different in that the distance between an object and a query is always the shortest distance from the object to a boundary of the query. Whereas in [27] this distance is zero if the object is contained within the query[1]. To amortize the cost of *SafeDist* computation, we combine it with the evaluation of the object on the Q-index, i.e., we execute a combined range and a modified nearest-neighbor query. The modification is that the distance between an object and a query is taken to be the shortest distance to any boundary even if the object is contained in the query (normally this distance is taken to be zero for nearest-neighbor queries). The combined search executes both

[1]Please note that in [27] the role of objects and queries is not reversed as it is here.

queries in parallel thereby avoiding repeated retrieval of the same nodes. *SafeSphere* is simply a circle centered at the current location of the object with a radius equal to *SafeDist*.

Given an object and a set of query rectangles, there exist various methods for determining safe rectangles. The related problem of finding a largest empty rectangle has been studied extensively and solutions vary from $O(n)$ to $O(n \log^3 n)$ time, (where n is the number of query rectangles) depending on restrictions on the regions [4, 5, 6, 22]. For our application, finding the "best", or maximal rectangle is not important for correctness (any empty rectangle is useful), we use a simple $O(n^2)$ time implementation for computing a safe rectangle. The implementation allows adaptations leading to approximations for the largest empty rectangle. The algorithm for finding the *SafeRect* for object O is as follows:

1. If object O is contained in a query, choose one such query rectangle and determine the relevant intersecting or contained query rectangles. If object O is not contained in a query rectangle, we consider all query rectangles as relevant. Let E be the set of relevant query rectangles.

2. Take object O as the origin and determine which relevant rectangles lie in which of the four induced quadrants. For each quadrant, sort the corner vertices of query rectangles that fall into this quadrant. For each quadrant determine the dominating points [10].

3. The dominating points create a staircase for each quadrant. Use the staircases to find the empty rectangle with the maximum area (using the property that a largest empty rectangle touches at least one corner of the four staircases).

We investigated several variations of this algorithm for safe rectangle generation. Variations include determining a largest rectangle using only a subset of the query rectangles, to determine relevant rectangles and limiting the number of combinations of corner points considered in the staircases. In order to determine a good subset of query rectangles we use the available *SafeDist*-value in a dynamic way. The experimental work for safe rectangle computations are based on generating safe rectangles which consider only query rectangles in a region that is ten times the size of *SafeDist*.

5 Velocity Constrained Indexing

In this section we present a second technique that avoids the two problems of traditional object indexing (viz. the high cost of keeping an object index updated as objects move and the need to reevaluate all queries whenever an object moves). The key idea is to avoid the need for continuous updates to an index on moving objects by relying on the notion of a maximum speed for each object. Under this model, an object will never move faster than its maximum speed. We term this approach, *Velocity Constrained Indexing* or *VCI*.

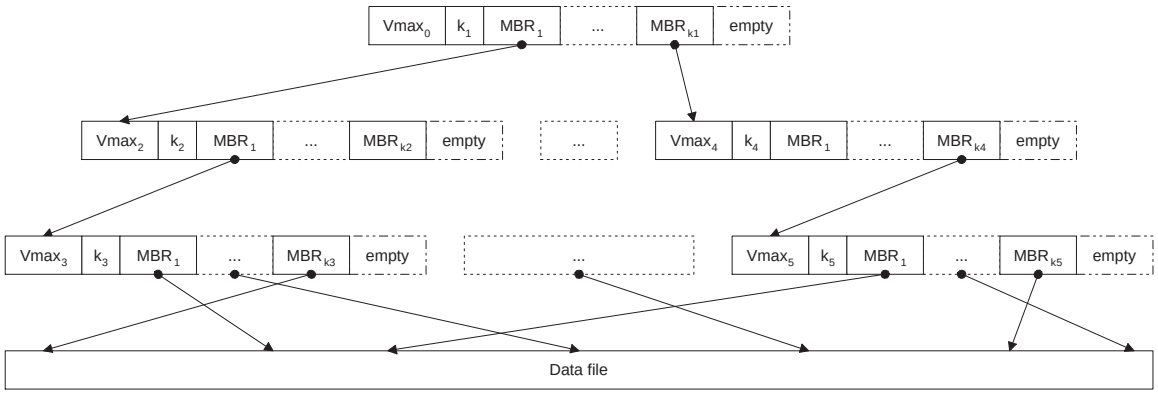
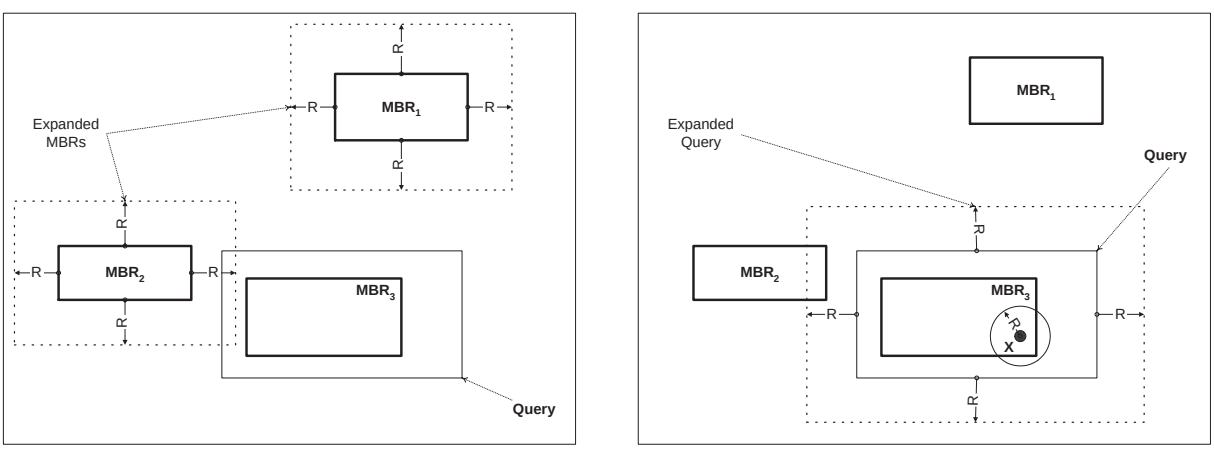


Figure 3: Example of Velocity Constrained Index (VCI)

A VCI is a regular R-tree based index on moving objects with an additional field in each node: v_{max}. This field stores the maximum allowed speed over all objects covered by that node in the index. The v_{max} entry for an internal node is simply the maximum of the v_{max} entries of its children. The v_{max} entry for a leaf node is the maximum allowed speed among the objects pointed to by the node. Figure 3 shows an example of a VCI. The v_{max} entry in each node is maintained in a manner similar to the MBRs of each entry in the node, except that there is only one v_{max} entry per node as compared to an MBR per entry of the node. When a node is split, the v_{max} for each of the new nodes is copied from the original node.

Consider a VCI that is constructed at time t_0. At this time it accurately reflects the locations of all objects. At a later time t, the same index does not accurately capture the correct locations of points since they may have moved arbitrarily. Normally the index needs to be updated to be useful. However, the v_{max} fields enable us to use this old index without updating it. We can safely assert that no point will have moved by a distance larger than $R = v_{max}(t - t_0)$. If we expand each MBR by this amount in all directions, the expanded MBRs will correctly enclose all underlying objects. Therefore, in order to process a query at time t, we can use the VCI created at time t_0 without being updated, by simply comparing the query with expanded version of the MBRs saved in VCI. At the leaf level, each point object is replaced by a square region of side $2R$ for comparison with the query rectangle[2].

An example of the use of the VCI is shown in Figure 4(a) which shows how each of the MBRs in the same index node are expanded and compared with the query. The expanded MBR captures the worst-case possibility that an object that was at the boundary of the MBR at t_0 has moved out of the MBR region by the largest possible distance. Since we are storing a single v_{max} value for all entries in the node, we expand each MBR by the same distance, $R = v_{max}(t - t_0)$. If the expanded MBR intersects with the query, the corresponding child is searched. Thus to process a node we need to expand all the MBRs stored in the node (except those that intersect without expansion, e.g. MBR_3 in Figure 4). Alternatively, we could perform a single expansion of the query by R and compare it with the unexpanded MBRs. An MBR will intersect with the expanded query if and only if the same MBR after expansion intersects with the original query. Figure 4 (b) shows the earlier example with query expansion. Expanding the query once per node saves some unnecessary computation.



(a) (b)

Figure 4: Query Processing with Velocity Constrained Index (VCI)

The set of objects found to be in the range of the query based upon an old VCI is a superset, S' of the exact set of objects that currently are in the query's range. Clearly, there can be no false dismissals in this approach. In order to eliminate the false positives, it is necessary to determine the current positions of all objects in S'. This can be achieved through a post-processing step. The current location of the object is retrieved from disk and compared with the query to determine the current matching. Note that it is not always necessary to determine the current location of each object that falls within

[2]Note that it should actually be replaced by a circle, but the rectangle is easier to handle.

the expanded query. From the position recorded in the leaf entry for an object, it can move by at most R. Thus its current location may be anywhere within a circle of radius R centered at the position recorded in the leaf. If this circle is entirely contained within the unexpanded query, there is no need to post-process this object for that query. Object X in Figure 4(b) is an example of such a point.

It should be noted that although the expansion of MBRs in VCI and the time-evolving MBRs proposed in [29] are similar techniques, the two are quite different in terms of indexing of moving objects. A key difference between the two is the model of object movement. Saltenis et al. [29] assume that objects report their movement in terms of velocities (i.e. an object will move with *fixed speed* in a *fixed direction* for a period of time). In our model the only assumption is that an object cannot travel faster than a certain known velocity. In fact, for our model the actual movement of objects is unimportant (as long as the maximum velocity is not exceeded). The time varying MBRs [29] exactly enclose the points as they move, whereas VCI pessimistically enlarges the MBRs to guarantee enclosure of the underlying points. Thus VCI requires no updates to the index as objects move, but post-processing is necessary to take into account actual object movement. The actual movement of objects has no impact on VCI or the cost of post-processing. Of course, as time passes, the amount of expansion increases and more post-processing is required.

Clustered VCI To avoid performing an I/O operation for each object that matches each expanded query, it is important to handle the post-processing carefully. We can begin by first pre-processing all the queries on the index to identify the set of objects that need to be retrieved for any query. These objects are then retrieved only once and checked against all queries. This eliminates the need to retrieve the same object more than once. We could still retrieve the same page containing several objects multiple times. To avoid multiple retrievals of a page, the objects to be retrieved can first be sorted on page number. Alternatively, we can build a clustered index. Clustering may reduce the total number of pages to be retrieved. We use the clustering option: i.e. the order of objects in the file storing their locations is organized according to the order of entries in the leaves of the VCI. Clustering can be achieved efficiently following creation of the index. A depth first traversal of the index is made and each object is copied from the original location file to a new file in the sequential order and the index pointer is appropriately adjusted to point to the newly created file. By default the index is not clustered. As is seen in Section 6, clustering the index improves the performance by roughly a factor of 3.

Refresh and Rebuild The amount of expansion needed during query evaluation depends upon two factors: the maximum speed v_{max} of the node, and the time that has elapsed since the index was created, $(t - t_0)$. Thus over time the MBRs get larger, encompassing more and more dead space, and may not be minimal. Consequently, as the index gets older its quality gets poorer. Therefore, periodically, it is necessary to *rebuild* the index. This essentially resets the creation time, and generates a index reflecting the changed positions of the objects. Rebuilding is an expensive operation and cannot be performed too often. A cheaper alternative to rebuilding the index is to *refresh* it. Refreshing simply updates the locations of objects to the current values and adjusts the MBRs so that they are minimal. Following refresh, the index can be treated as though it has been rebuilt.

Refreshing can be achieved efficiently by performing a depth first traversal of the index. For each entry in a leaf node the latest location of the object is retrieved (sequential I/O if the index is clustered). The new location is recorded in the leaf page entry. When all the entries in a leaf node are updated, we compute the MBR for the node and record it in the parent node. For directory nodes when all MBRs of its children have been adjusted, we compute the overall MBR for the node and record it in the parent. This is very efficient with the depth first traversal. Although refresh is more efficient than a rebuild, it suffers from not altering the structure of the index – it retains the earlier structure. If points have moved significantly, they may better fit under other nodes in the index. Thus there is a trade-off between the speed of refresh and the quality of the index. An effective solution is to apply several refreshes followed by a less frequent rebuild. Experimentally, we found that refreshing works very well.

6 Experimental Evaluation

In this section we present the performance of the new indexing techniques and compare them to existing techniques. The experiments reported are for two-dimensional data, however, the techniques are not limited to two dimensions. The various indexing techniques were implemented as R^*-trees [9] and tested on synthetic data. The dataset used consists of 100,000 objects composed of a collection of 5 normal distributions each with 20,000 objects. The mean values for the normal distribution are uniformly distributed, and the standard deviation is 0.05 (the points are all in the unit square). The centers of queries are also assumed to follow the same distribution but with a standard deviation of 0.1 or 1.0. The total number of queries is varied between 1 and 10,000 in our experimentation. Each query is a square of side 0.01. Other experiments with different query sizes were also conducted but since the results are found to be insensitive to the query size, they are not presented. More important than query size is the total number of objects that are covered by the queries and the number of queries.

The maximum velocities of objects follow a Zipf distribution with an overall maximum value of V_{max}. For most experiments V_{max} was set to 0.00007 – if we assume that the data space represents a square of size 1000 miles (as in [18]), this corresponds to an overall maximum velocity of 250 miles an hour. In each experiment, we fix the fraction of objects, m, that move at each time step. This parameter was varied between 1000 and 10,000. The time step is taken to be 50 seconds. At each time step, we randomly select m objects and for each object, we move it with a velocity between 0 and the maximum velocity of the object in a random direction. The page size was set to 2048 bytes for each experiment. As is customary, we use the number of I/O requests as a metric for performance. The top two levels of each index were assumed to be in memory and are not counted towards the I/O cost. The various parameters used are summarized in Table 1.

Parameter	*Meaning*	*Values*
N	Number of Objects	100,000
m	Number of objects that move at each time step	1000 – 10,000
q	Number of queries	1 – 10,000
V_{max}	Overall maximum speed for any object	50mph, 125mph, 250mph, 500mph

Table 1: Parameters used in the experiments

6.1 Traditional Schemes

We begin with an evaluation of *Brute Force* and traditional indexing. Updating the index to reflect the movement of objects can be achieved using several techniques:

1. *Insert/Delete*: each object that moves is first deleted and then re-inserted into the index with its new location.

2. *Reconstruct*: the entire index structure can be recomputed at each time step.

3. *Modify*: the positions of the objects that move during each time step are updated in the index.

The *modify* approach is similar to the technique for handling movement of points proposed by Saltenis et. al. [29] wherein the bounding boxes of the nodes are expanded to accommodate the past, current, and possibly future positions of objects. The *modify* approach differs from these because it does not save past or future positions in the index which is acceptable since the purpose of this index is primarily to answer continuous queries based upon the current locations of the objects. The approach of [29] assumes that objects move in a straight line with a fixed speed most of the time. Whenever the object's speed or velocity changes, an update is sent. The index is built using this speed information. Their experimental results are based upon objects moving between cities which are assumed to be connected by straight roads and the objects moves with

a very regular behavior – for the first sixth of a route they accelerate at a constant rate to one of three maximum velocities which they maintain until the last sixth of the route at which point they decelerate. Our model for object movement is more general and does not require that object maintain a given velocity for any point in time. Under this model, the approach of [29] is not applicable.

Table 2 shows the relative performance of these schemes in terms of number of I/O operations performed. The performance of these approaches does not vary over time, hence we simply report a single value for each combination of m and q. We assume that the top two levels of the index are memory resident. For these experiments that roughly corresponds to about 21 pages in memory. The I/O numbers for *Brute Force* are evaluated assuming efficient use of 21 buffers: 20 buffers are assumed to hold blocks of queries.

Parameters		*Number of I/O Operations*			
m	q	*Reconstruct*	*Insert/Delete*	*Modify*	*Brute Force*
1,000	1,000	211,817	5,865	3,806	1,010
1,000	10,000	228,308	22,356	20,298	5,100
10,000	1,000	211,817	43,413	22,581	1,010
10,000	10,000	228,308	59,904	39,072	5,100

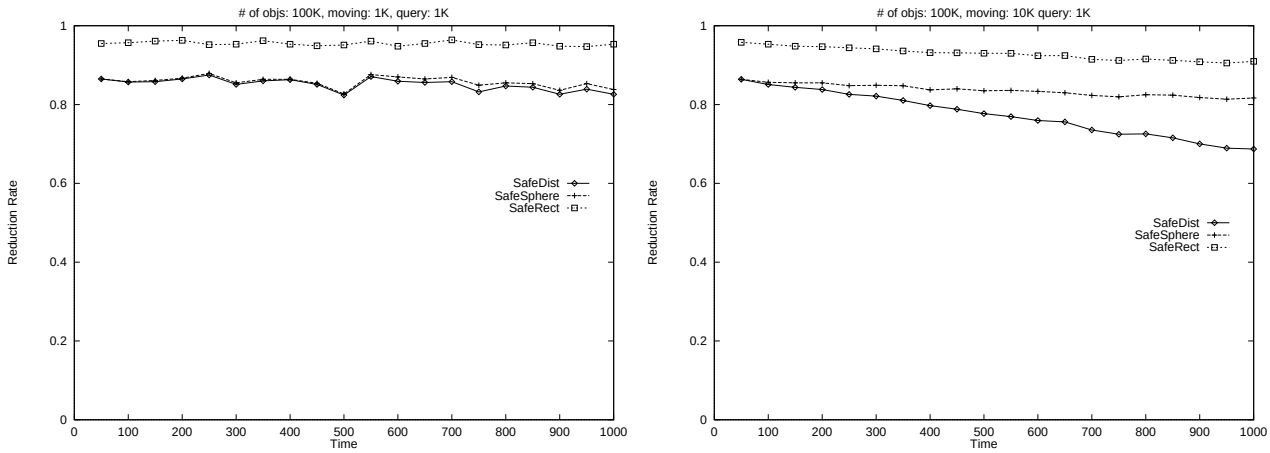
Table 2: Performance of traditional techniques.

From the table it is clear that the *Brute Force* approach gives the best performance in all cases. This is largely due to the fact that this approach does not need to maintain any structures as objects move. We assume that there are enough buffers to hold only the first two levels of the other indexes when computing brute force. The *Reconstruct* approach is clearly the poorest since it is too expensive to build the index at each time step. The *Insert/Delete* scheme incurs roughly double the I/O cost that *Modify* incurs to update the index while their query cost is the same.

We point out that while the *Brute Force* approach has the lowest I/O cost, it may not be the best choice. The reason is that unlike the other schemes which employ an index on the objects to significantly reduce the number of comparisons needed, *Brute Force* must compare each object with each query. Thus it is typically going to incur almost three orders of magnitude more comparisons than the others! An earlier experiment to measure the total time required for *Modify* and *Brute Force* showed that their performance is very comparable despite the low I/O cost of *Brute Force* [7]. Except for this special case, the I/O cost is a good measure of performance.

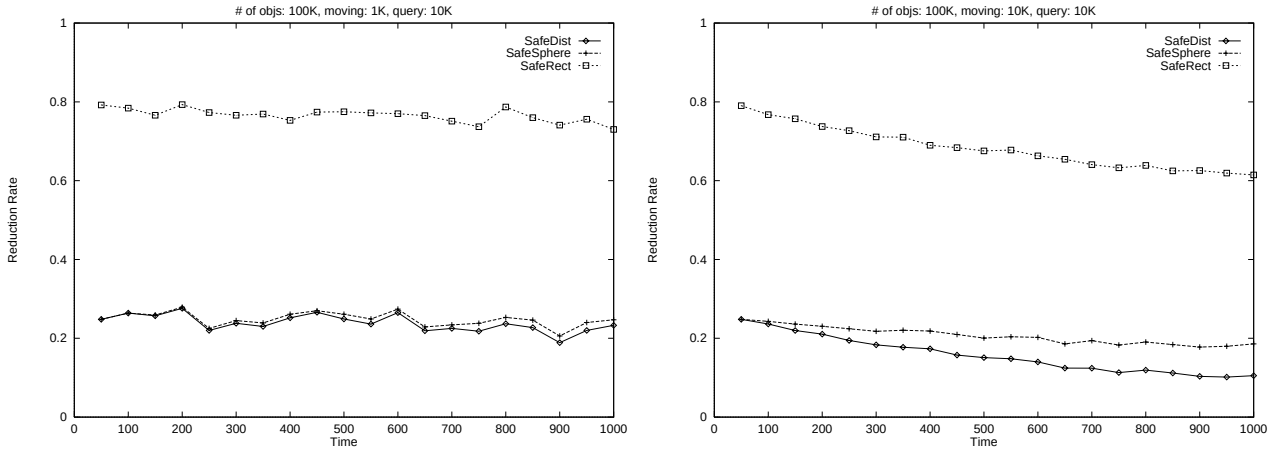
6.2 *Safe Region* Optimizations

We now study the performance of the safe region optimizations among themselves. For each scheme we plot the *Reduction Rate*: the fraction of moved objects that are within their safe region. These objects do not need to report their location. We study the effectiveness of each measure as time passes. Figures 5 and 6 show the results for various combinations of m and q. For example, Figure 5(b) shows the reduction rates for 10% objects moving at each time step, and 1000 queries. Each of the safe regions is computed at time 0. As long as the object is within its safe region it does not report its new location. As expected, the fraction of objects that remain within their safe regions drops as time passes (shown along the x-axis). For example after 100 time steps 95% of the objects are still within their *SafeRect* and need not report their positions, whereas 83% of the objects are within their *SafeDist*. Thus *SafeRect* is more effective in reducing the need for objects reporting their locations. An important point to note is that even though we do not re-compute the safe regions in our experiments, we find that the safe region optimizations remain very effective for large durations. This is important since the cost of computing these measures is high. The cost of computing *SafeDist* and *SafeSphere* is on the order of 13 I/O operations per object for the case of 10,000 queries. The cost of computing *SafeRect* is significantly higher – around 52 I/Os per object. These high costs do not adversely affect the gains from these optimizations since the re-computations can be done infrequently.



(a) (b)

Figure 5: Safe Region Optimizations with (a) 1% moving and 1% queries, and (b) 10% moving and 1% queries



(a) (b)

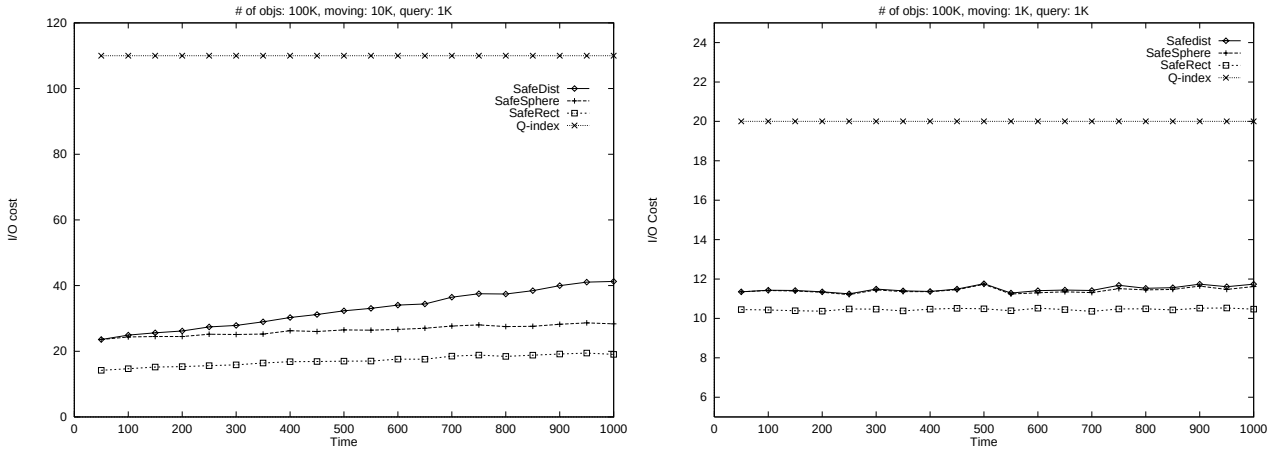
Figure 6: Safe Region Optimizations with (a) 1% moving and 10% queries, and (b) 10% moving and 10% queries

A common trend across all graphs is that the *SafeRect* measure is most effective. *SafeSphere* is never worse in performance than *SafeDist*. This is not surprising since *SafeSphere* augments *SafeDist* with the center information to provide a safe region as opposed to an absolute measure of movement. Thus under *SafeSphere* an object can re-enter the safe region whereas an object that has moved by *SafeDist* must always be tested against Q-index until *SafeDist* is re-computed. To see why *SafeRect* outperforms *SafeSphere*, consider that the *SafeSphere* pessimistically limits the region of safety in all directions by the shortest distance from the object to a query boundary. On the other hand, *SafeRect* selects four distances, one in each direction, to the nearest query boundary. Thus it is more likely to extend further than the *SafeSphere* [3]. This is especially the case when the number of queries gets very large, as is seen from Figures 6 (a) and 6 (b) where *SafeRect* performs significantly better than the other two optimizations as we go from 1000 to 10000 queries. The optimizations remain effective even when the number of moving objects is increased ten-fold to 10,000, as is seen from the similarity between Figure 5(a) and Figure 5(b) (also between Figure 6 (a) and Figure 6 (b)).

[3] It should be noted that *SafeRect* can be more constraining than *SafeSphere* in one or more directions.

6.3 Incremental Evaluation and the Q-index Approach

We now compare the performance of the Q-index approach with the traditional approaches. Let us first assume that there is enough memory available to keep Q-index entirely memory resident. Under this assumption at each time step, we simply need to read in the positions of only those objects that have moved since the previous evaluation. A separate file containing only these points can be easily generated on the server during the period between evaluations. Thus the I/O cost of the Q-index approach is simply given by the number of pages that make up this file. The safe region optimizations further reduce the need for I/O by effectively reducing the number of objects that report their updates. Figures 7 through 8 show the performance of the various schemes. We do not present the traditional schemes due to the performance being almost two orders of magnitude worse (c.f. Table 2).



(a) (b)

Figure 7: Performance of the Q-index techniques with (a) 10% moving and 1% queries, (b) 1% moving and 1% queries

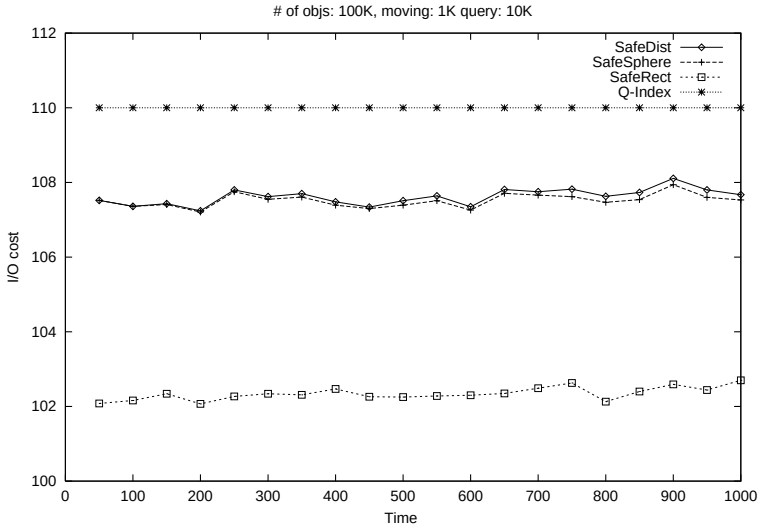


Figure 8: Performance of the Q-index techniques with 1% moving and 10% queries with memory-resident Q-index

In Figure 7(a) the results with 1000 queries and 10,000 objects moving at each time step are shown. The Q-index approach requires 110 pages of I/O at each time step to retrieve the new locations of the moved objects and process them

against the memory-resident Q-index. It should be pointed out that this is actually sequential I/O. This is a significant reduction from the I/O cost of the traditional approaches as shown in Table 2 representing more than an order of magnitude improvement. The optimizations further reduce the I/O cost by almost another order of magnitude. Of course, this reduction reduces over time but not significantly even for as many as 1000 time steps.

For smaller numbers of moving objects, the Q-index needs to perform proportionately smaller numbers of I/O operations. This can be seen from Figure 7 (b) where only 20 I/Os are needed at each time step for Q-index. As the number of objects that move at each time step in increased, the Q-index approach needs to perform increased I/O until eventually a sequential scan of the entire data file is required when virtually every object moves in each time step. Thus the approach scales well with the movement of objects, gracefully degrading to a sequential scan.

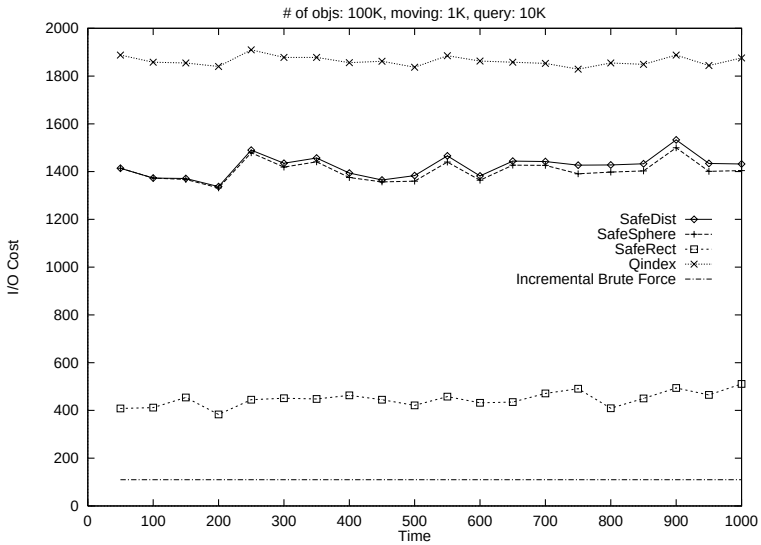


Figure 9: Performance of Q-index techniques with 1% moving and 10% queries

The above experiments were conducted with 1000 concurrent queries. If the number of queries is smaller, Q-index will fit in memory and the I/O costs will be largely unchanged. However, for larger numbers of queries it is possible that the entire index does not fit in main memory, possibly resulting in page I/O for each object that is queried. If the number of objects that need to be queried against the Q-index is large, this may significantly increase the amount of I/O. Figure 9 shows the performance of 10,000 queries with 10,000 objects moving at each time step under the assumption that only top two levels of Q-index are in memory. In comparison to Figure 8, which represents the assumption that the index is memory-resident, index I/O exacts a high price. It should be pointed out, however, that even with this very large increase in I/O, the Q-index approach is still superior to the traditional approaches. For example, the I/O cost of *Modify* with the two settings of m=1,000, and q=10,000 is 20,298, and that of *Brute Force* is 5,100.

If we consider only I/O for *Brute Force*, an incremental version can outperform the Q-index based approach if the Q-index does not fit in memory. Consider the above case with 10,000 queries and 1000 objects moving at each time step. This corresponds to 100 pages for queries and 10 pages for objects that move at each time. If we assume that $B+1$ buffers are available, then *brute force* can evaluate all queries with $\lceil \frac{100}{B} \rceil (B+10)$ I/O operations. With as few as 21 buffers, this requires only 150 I/Os as compared to over 1800 for Q-index! However, as pointed out earlier, the *Brute Force* approach pays a high computation price that offsets the reduced I/O. To validate this claim, we conducted an experiment where we measured the total time taken (in seconds) by *Brute Force* and the other proposed approaches. The results are shown in Table 3 for two sets of values of m and q. We see that although *Brute Force* would have only a fraction of the I/O operations required by the others, it is actually slower overall. We again point out that this anomaly of I/O time not translating to overall performing happens only for *Brute Force* due to its inordinately large numbers of comparisons.

m	*q*	*Brute Force*	*Q-index*	*SafeSphere*	*SafeRect*
1000	10,000	3.6s	1.7s	0.9s	0.5s
10,000	10,000	37s	3.1s	1.3s	1.1s

Table 3: Impact of CPU cost

Impact of Velocity. From the above experiments we find that the incremental Q-index approach and optimizations scale well with variations in the number of moving objects and queries. To study the impact of the degree of movement of the objects, we conducted two experiments where we altered the speed distributions. Figure 10 shows the performance with the maximum allowable speed for any object, V_{max}, reduced by a factor of two. The results with a V_{max} corresponding to 500 miles per hour is very similar except that the optimizations are a little less effective. The relative performance of the newly proposed schemes is unaffected by these changes in the allowable speeds of objects. We see only a slight change in the effectiveness of the optimizations.

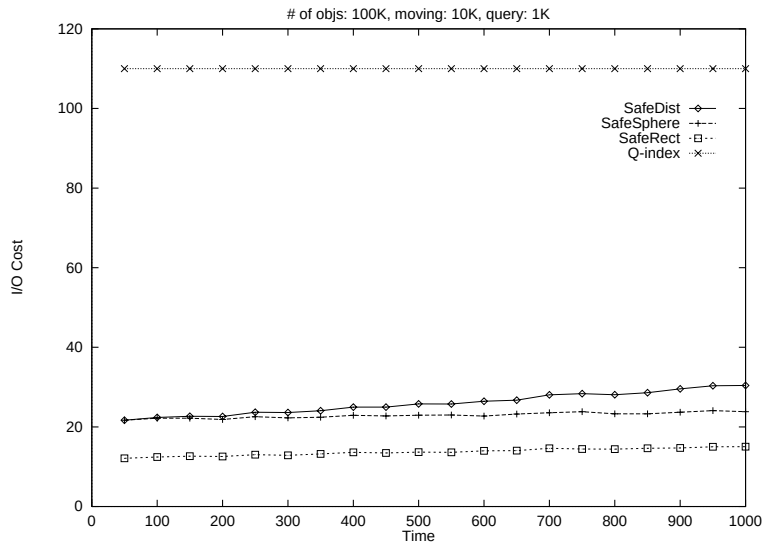


Figure 10: Q-index techniques with 10% moving and 1% queries and V_{max}=125mph

Denser Object Locations. We now investigate the effect of a much denser set of objects and queries corresponding to a smaller region such as a city. In this experiment the range of the space is reduced from 1000 miles by 1000 miles to a 10 mile by 10 mile region. The number of objects is maintained at 100,000 with 1000 moving at each time step. The number of queries is 10,000. Since a city is likely to have a more uniform distibution, we increased the standard deviation to 0.8. Also the maximum speed is reduced from 250 miles per hour to 50 miles per hour. The performance for these settings is shown in Figure 11. In comparison with the wider area setting (c.f. Fig 8) we can see that the performance is poorer by about a factor of 10. This reduction in performance is not surprising since the safe region optimizations are less effective. This is directly related to the rate at which objects exit their safe regions. The important point however, is that the Q-Index approaches are still an order of magnitude better than the traditional approaches.

6.4 Velocity Constrained Indexing

Next we discuss the performance of the Velocity Constrained Indexing (VCI) technique. There are two components of the cost for VCI: i) pre-processing to evaluate the expanded queries on VCI; and ii) post-processing to eliminate false positives. Since the VCI approach is unaffected by the actual number of objects that move at each time step (i.e., *m*), all objects were

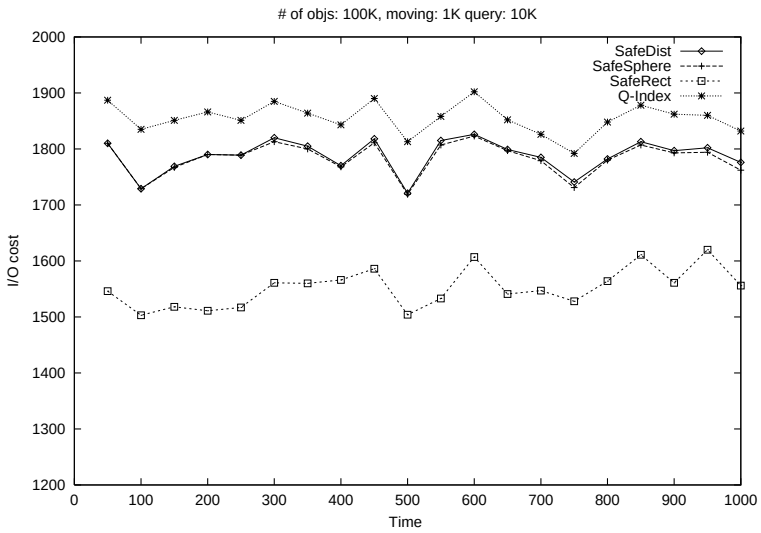
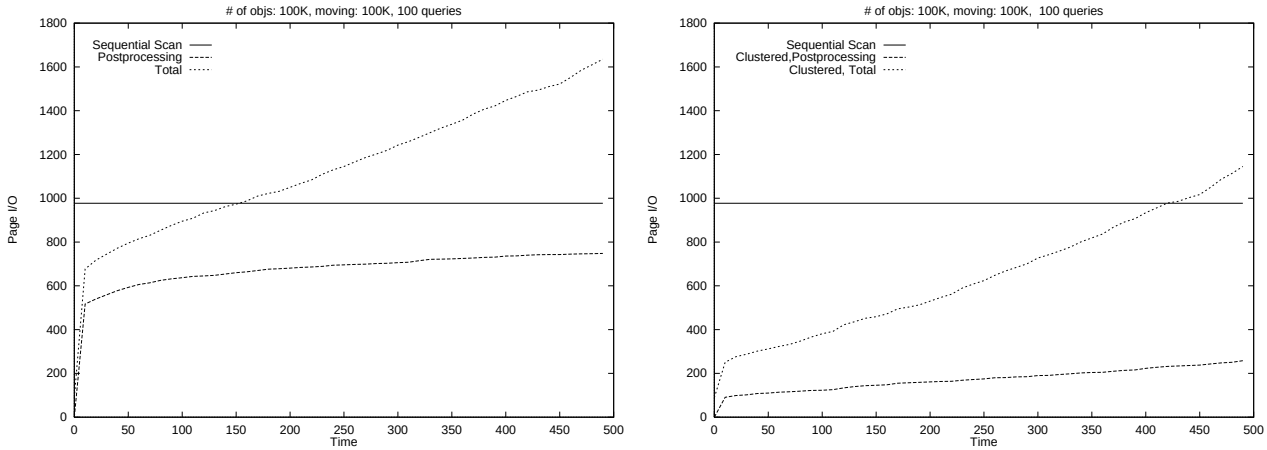


Figure 11: Q-index techniques for Dense data with 1% moving and 10% queries

moved at each time step. Figure 12(a) shows the performance of VCI for 100,000 objects moving at each time instant, and 100 queries. The pre-processing cost increases with time since the queries get larger due to greater expansion resulting in more parallel path searches on the VCI. Similarly, post-processing cost increases with time since more and more false positives are likely to be found with increased query expansions. The graph shows the post-processing cost and the total cost as time since creation of VCI. The cost of a sequential scan of the entire object file is also shown.



(a) (b)

Figure 12: Performance of Velocity Constrained Indexing (a) No Clustering; (b) With Clustering

The total cost approaches that of a sequential scan after about 150 time steps, at which point the VCI is not effective – it would be more efficient to scan the file instead of using the index. Figure 12(b) shows the improvement due to clustering. There is a very significant improvement in post-processing cost resulting in about 400 fewer I/O operation at each time step. Thus clustering extends the utility of the VCI from about 150 time steps to over 400 time steps.

Refresh and Rebuild. Figure 13 shows the impact of applying a *refresh* to the VCI. The refresh helps reduce both the pre- and post-processing costs. The pre-processing cost is reduced since the MBRs better fit the underlying data and the clock for query expansion is reset. This improves the quality of the index resulting in faster query processing. The post-processing

cost is reduced since there will be fewer false positives as a result of a "tighter" index. The overall cost is reduced by almost 600 I/O operations immediately following the refresh. Over time it again degrades and another refresh is applied, etc. The refresh period can be adjusted as necessary. In this experiment, the refresh was performed to keep the total cost below that of a sequential scan. The application of a rebuild has a very similar effect to that of a refresh. The difference would show up only for very large time intervals when objects have moved so much that the old VCI organization is inefficient. The effect of rebuilding would be very similar to that of running the test again from time step 0, hence we do not consider it here.

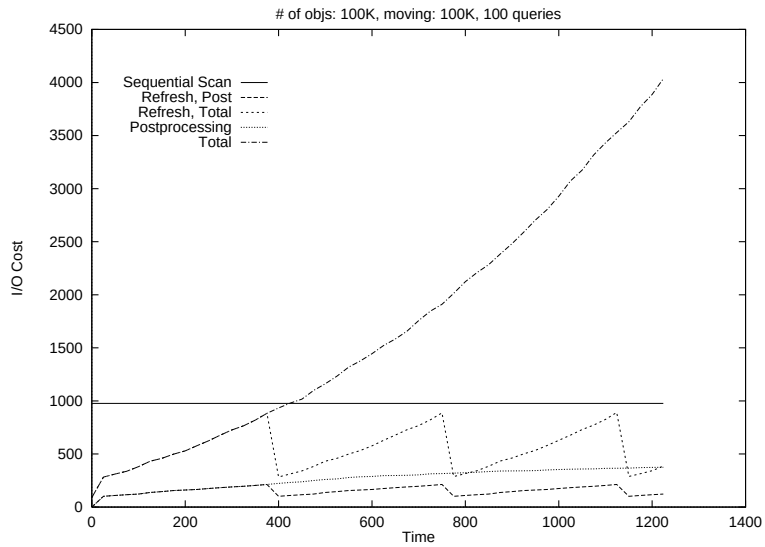


Figure 13: Impact of Refresh on Velocity Constrained Indexing

Sensitivity to Parameters. The VCI approach is not affected by the actual movement of objects (other than through the maximum speeds). Thus the costs would not change even if all the objects were moving at each time instant! On the other hand, the VCI approach is very sensitive to the number of queries. The above graphs are for only 100 queries. If the number of queries is increased to 1000, we find that the pre-processing cost increases proportionately, as does the post-processing cost. Very soon after creation of the VCI its performance degrades to worse than a sequential scan forcing frequent refreshing. This impacts the performance and renders the scheme unusable. Thus VCI is a reasonable approach when the queries cover a small number of objects. With 10 queries, we find that the graph scales down roughly linearly too, e.g. for a single query the post-processing and total costs are 6, and 17 I/Os, respectively. To study the impact of denser distributions of queries, we repeated the above tests for the VCI approach with a query set having one-tenth the standard deviation of the other tests (viz. 0.1). The results are shown in Figure 14. The relative performance of the graphs is very similar to that seen with a broader distribution, except that the degradation towards a sequential scan occurs much faster. This is expected since each query now covers more objects on the average.

The maximum speed of objects is clearly an important parameter for the VCI approach. To study the impact of V_{max} (the overall maximum speed of any object) on performance we conducted several experiments with different values of V_{max}. The results resemble very closely those obtained earlier for VCI. The major impact of changes in maximum speed is that the time scale get stretched (contracted) if V_{max} is reduced (increased). The stretching is linearly related to the changes in speed. Other than this difference, the graphs are very similar. This is not surprising because the important factor in determining the performance of VCI is the amount of expansion that a query experiences: $R = v_{max}(t - t_0)$ (v_{max} is the maximum velocity field stored in the node being examined). With double the speed, we need half the time difference to achieve the same expansion. Therefore if the max speed in increased by a large factor, such as with the experiment on the 10 miles by 10 miles range (this is effectively increasing the speed of the objects) VCI becomes quite ineffective. In fact with 10 queries, a sequential scan would be better after only 45 seconds.

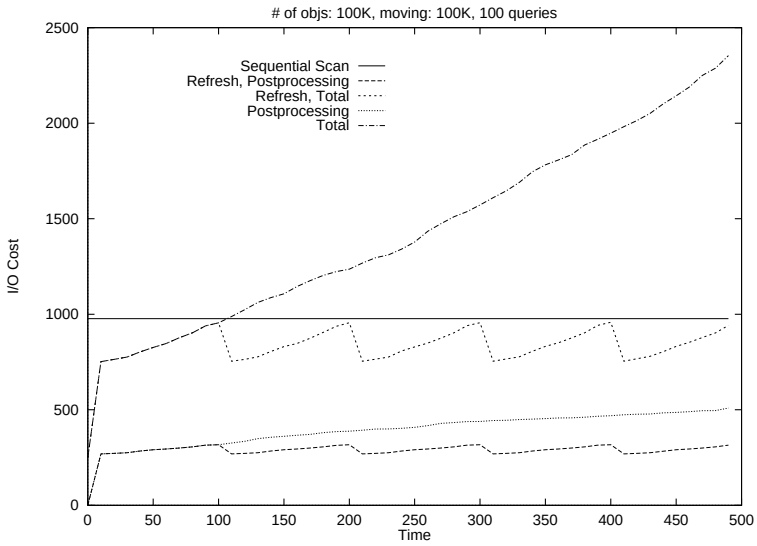


Figure 14: Performance of Velocity Constrained Indexing with query std = 0.1

Comparison to Q-index. Our experimental work indicates that the Q-index approach outperforms the VCI approach. For even a hundred queries, VCI incurs between 280 and 880 I/O operations (Figure 12). For larger numbers of queries, it will certainly not incur any less since each extra query will add to the query processing cost as well as potentially generate new objects that need to be post-processed. In contrast, for 1000 queries, Q-index needs 110 I/Os without safe region optimizations, and only about 20 I/Os with the *SafeRect* optimization. A positive aspect of VCI is that it is insensitive to variations in the number of moving objects, *m*. Even if all the objects move, the Q-index approach[4] will incur a sequential scan. Thus it is possible that for very few queries and very large numbers of objects moving at each time instant, VCI could outperform Q-index, however this is not very practical.

The key advantage of (and also the motivation for developing) Velocity Constrained Indexing is its ability to handle arbitrary changes to the set of continuous queries. The Q-index approach is forced to make a sequential scan of the entire set of objects for each newly arriving query (although queries that arrive within a single time step can be handled with a single scan).

6.5 Combined Indexing Scheme

The results show that query indexing and safe region optimizations significantly outperform the traditional indexing approaches and also the VCI approach. These improvements in performance are achieved by eliminating the need to evaluate all objects at each time step through incremental evaluation. Thus they perform well when there is little change in the queries being evaluated. The deletion of queries can be easily handled simply by ignoring the deletion until the query can be removed from the Q-index. The deleted query may be unnecessarily reducing the safe region for some objects, but this does not lead to incorrect processing and the correct safe regions can be recomputed in a lazy manner without a significant impact on the overall costs.

The arrival of new queries, however, is expensive under the query indexing approach as each new query must initially be compared to every object. Therefore a sequential scan of the entire object file is needed at each time step that a new query is received. Furthermore, a new query potentially invalidates the safe regions rendering the optimizations ineffective until the safe regions are recomputed. The VCI approach, on the other hand, is unaffected by the arrival of new queries (only the total number of queries being processed through VCI is important). Therefore to achieve scalability under the insertion and

[4] Assuming there is enough memory to hold the queries – which is also assumed by the VCI approach since it only handles small numbers of queries.

deletion of queries we propose a *combined scheme*. Under this scheme, both a Q-Index and a Velocity Constrained Index are maintained. Continuous queries are evaluated incrementally using the Q-index and the *SafeRect* optimization. The Velocity Constrained Index is periodically refreshed, and less periodically rebuilt (e.g. when the refresh is ineffective in reducing the cost). New queries are processed using the VCI. At an appropriate time (e.g. when the number of queries being handled by VCI becomes large) all the queries being processed through VCI are transferred to the Query Index in a single step. As long as not too many new queries arrive at a given time (e.g. less than 10 in each time step), this solution offers scalable performance that is orders of magnitude better than the traditional approaches.

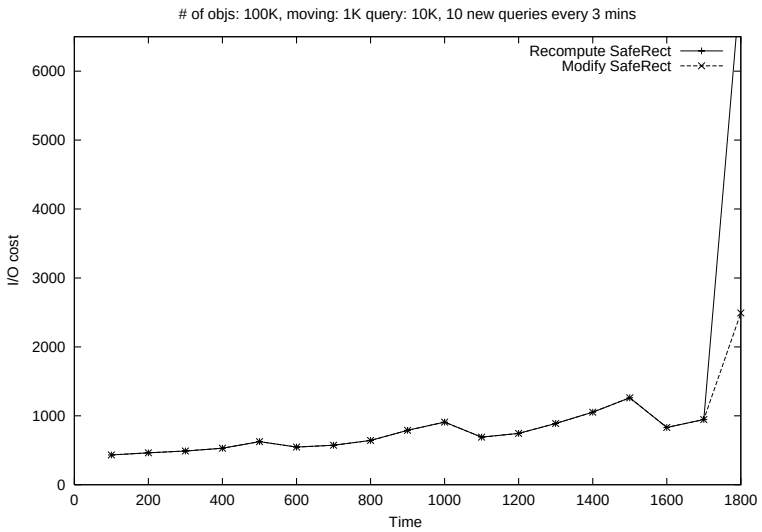


Figure 15: Performance of VCI and Q-Index With Dynamic Queries

We now present the performance of the combined scheme. The experiment is conducted with 100,000 objects with 1% moving. The experiment begins with 10,000 queries and new queries arrive at the rate of 10 queries every three minutes (actually one query every 18 seconds). Newly arriving queries are handled by the VCI index while the ongoing queries are processed using the Q-Index. When the number of queries handled by the VCI index reaches a threshold (100 in this experiment), we ingest the 100 queries into the Q-Index in a single step. This ingestion requires changes to the index structure and also potentially changes the safe regions for the objects. We consider two approaches for correcting the safe regions: recomputing all the safe regions, and modifying the safe regions by comparing them against the ingested queries. Figure 15 shows the combined cost of the two indexes over time as objects arrive. Only the SafeRect region is considered. As can be seen from the graph, the combined cost remains very small until the point at which the queries are ingested. At this time, a large penaltly is paid for computing the new SafeRect regions. The recompute approach is very expensive, however the modify approach does not incur a very large overhead. The effect of three refreshes on the VCI is clearly visible.

It should be noted that since the newly arriving queries are incorporated into the Q-Index periodically, it is not neccessary that all incoming queries need to be immediately handled by VCI. In fact, only urgent queries need to be handled by VCI, others can begin their evaluation only after the next time that queries are ingested into Q-Index. From the results we observe that with this combined approach the overall performance is still much better than the traditional approaches.

7 Conclusion

Moving object environments are characterized by large numbers of moving objects and concurrent active queries over these objects. Efficient continuous evaluation of these queries in response to the movement of the objects is critical for supporting

acceptable response times. We showed that the traditional approach of building an index on the objects (data) can result in poor performance. In fact, a brute force, no index strategy gives better performance in many cases. Neither the traditional approach, nor the brute force strategy achieve reasonable performance.

We presented two novel indexing techniques for scalable execution: *Query Indexing* and *Velocity Constrained Indexing (VCI)*. Our experimental results demonstrated that query indexing achieves very significant improvement over the traditional approaches (as much as two orders of magnitude), but does not efficiently handle the arrival of new queries. Although the VCI approach gives good performance only for small numbers of queries, it is unaffected by changes in queries and actual object movement. Thus we see that the two techniques complement each other enabling a combined solution that efficiently handles not only ongoing queries but also dynamically inserted queries. The experiments also demonstrated the robustness of the new techniques to variations in the parameters. The combined schemes therefore achieve superior performance to existing solutions for the efficient and scalable evaluation of continuous queries over moving objects.

References

[1] S. Acharya, M. J. Franklin, and S. Zdonik. Disseminating updates on broadcast disks. In T. M. Vijayaraman et al., editors, *Proceedings of the twenty-second international Conference on Very Large Data Bases, September 3–6, 1996, Mumbai (Bombay), India*, pages 354–365, Los Altos, CA 94022, USA, 1996. Morgan Kaufmann Publishers.

[2] Swarup Acharya, Rafael Alonso, Michael J. Franklin, and Stanley B. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 199–210, 22–25 May 1995.

[3] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. 2000 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Dallas, Texas, May 2000.

[4] A. Aggarwal and S. Suri. Fast algorithms for computing the largest empty rectangle. In *Proceedings of the 3rd Symposium on Computational Geometry*, pages 278–290, 1987.

[5] A. Aggarwal and J. Wein. Computational geometry. Lecture Notes for MIT, 1988.

[6] N. Amenta. Bounded boxes, hausdorff distance, and a new proof of an interesting helly-type theorem. In *Proceedings of Symp. on Computational Geometry*, pages 340–347, 1994.

[7] W.G. Aref, S.E. Hambrusch, and S. Prabhakar. Information management in a ubiquitous global positioning environment. Technical Report 00-006, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, February 2000.

[8] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, December 1996.

[9] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, May 23-25 1990.

[10] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[11] US Wireless Corp. The market porential of the wireless location indistry. http://www.uswcorp.com/USWCMainPages/laby.htm.

[12] L. Forlizzi, R. H. Guting, E. Nardelli, and M. Scheider. A data model and data structures for moving objects databases. In *Proc. of ACM SIGMOD Conf.*, Dallas, Texas, May 2000.

[13] R.H. Guting, M.H.Bohlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 2000. To Appear.

[14] S. E. Hambrusch, C.-M. Liu, W. Aref, and S. Prabhakar. Query processing in broadcasted spatial index trees. In *7th International Symposium on Spatial and Temporal Databases (SSTD 2001)*, July 2001.

[15] Q. Hu, W.-C. Lee, and D. L. Lee. Power conservative multi-attribute queries on data broadcast. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 157–166, 2000.

[16] Q. Hu, W.-C. Lee, and D. L. Lee. A hybrid index technique for power efficient data broadcast. *Distributed and Parallel Databases*, 9(2):151–177, 2001.

[17] Tomasz Imieliński, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the International Conference on Management of Data*, pages 25–36. ACM Press, May 1994.

[18] G. Kollios, D. Gunopulos, and V.J. Tsotras. On indexing mobile objects. In *Proc. 1999 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Philadelphia, June 1999.

[19] H. Koshima and J. Hoshen. Personal locator services emerge. *IEEE Spectrum*, 37(2):41–48, February 2000.

[20] Anil Kumar, Vassilis J. Tsotras, and Christos Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1–20, 1998.

[21] Trimble Navigation Ltd. Trimble customer solutions. http://www.trimble.com/solution/index.htm, 1999.

[22] M. McKenna, J. O'Rourke, and S. Suri. Finding the largest rectangle in an orthogonal polygon. In *Proceedings of the 23rd Allerton Conference on Communication, Control, and Computing*, pages 486–495, 1985.

[23] Rand McNally. Streetfinder GPS for palm IIIc connected organizer. http://www.randmcnally.com/palmIIIc/index.ehtml#receiver.

[24] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-objects representations. In *Proceedings of the SSDBM Conf.*, pages 123–132, 1999.

[25] D. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[26] D. Pfoser, Y. Theodoridis, and C.S. Jensen. Indexing trajectories of moving point objects. Technical Report CH-99-3, Chorochronos Tech. Rep., June 1999.

[27] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 71–79, San Jose, CA, 1995.

[28] S. Prabhakar S. Hambrusch, W. Aref. Pervasive location-aware computing enviroments. http://www.cs.purdue.edu/place.

[29] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the position of continuously moving objects. In *Proceedings of ACM SIGMOD Conference*, Dallas, Texas, May 2000.

[30] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[31] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE'97)*, pages 422–432, 1997.

[32] Jamel Tayeb, Özgür Ulusoy, and Ouri Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[33] TruePosition. What is trueposition cellular location system? http://www.trueposition.com/intro.htm.

[34] Jay Werb and Colin Lanzl. Designing a positioning system for finding things and people indoors. *IEEE Spectrum*, 35(9):71–78, September 1998.

[35] Ouri Wolfson. Research issues on moving object databases (tutorial). In *Proceedings of ACM SIGMOD Conference*, page 581, Dallas, Texas, May 2000.

[36] Ouri Wolfson, Sam Chamberlain, Son Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE'98)*, Orlando, FL, February 1998.

[37] Ouri Wolfson, Prasad A. Sistla, Sam Chamberlain, and Yelena Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[38] Ouri Wolfson, Bo Xu, Sam Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proceedings of the SSDBM Conf.*, pages 111–122, 1998.

[39] J. M. Zagami, S. A. Parl, J. J. Bussgang, and K. D. Melillo. Providing universal location services using a wireless e911 location network. *IEEE Communications Magazine*, April 1998.

[40] Stanley Zdonik, Michael Franklin, Rafael Alonso, and Swarup Acharya. Are "disks in the air" just pie in the sky? In *IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.