Main Memory Evaluation of Monitoring Queries Over Moving Objects [*]

Dmitri V. Kalashnikov Sunil Prabhakar Susanne E. Hambrusch
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907
{dvk,sunil,seh}@cs.purdue.edu

Abstract

In this paper we evaluate several in-memory algorithms for efficient and scalable processing of continuous range queries over collections of moving objects. Constant updates to the index are avoided by query indexing. No constraints are imposed on the speed or path of moving objects or fraction of objects that move at any moment in time. We present a detailed analysis of a grid approach which shows the best results for both skewed and uniform data. A sorting based optimization is developed for significantly improving the cache hit-rate. Experimental evaluation establishes that indexing queries using the grid index yields orders of magnitude better performance than other index structures such as R*-trees.

1 Introduction

The problem of handling different types of queries on moving objects has caught wide attention recently due to the development of location detection and wireless technologies [2][8][15][24]. Using these technologies, it is possible to develop systems where a local server tracks the locations of moving objects. Mobile objects can report their location to this server through a wireless interface, or the objects can be tracked through ground-based radar or satellites.

As an example, consider a system where aircraft are tracked by radars. Regions of space in which aircraft can be detected by enemy radars or anti-aircraft systems are identified by a server. The server continuously monitors the location of friendly aircraft with respect to these regions and issues alerts as soon as an aircraft is in a detection area. Alternatively, there might be areas where only specially designated aircraft are allowed to be. Such areas can be monitored continuously at the server to detect trespassers. More generally, the location of people and vehicles can be tracked with such systems.

In this paper we address the problem of evaluating multiple concurrent continuous range queries on moving objects. In contrast to regular queries that are evaluated once, a continuous query remains active over a period of time and has to be continuously evaluated during this time. Each of these queries needs to be re-evaluated as the objects move. A solution for continuous *real-time* evaluation of queries that scales to large numbers of queries and objects is a major challenge.

[*]Work Supported by NSF grants 9988339-CCR, 9972883, 0010044-CCR, and Career award 9985019-IIS and a Gift from Microsoft Corporation.

While the focus of our work is on moving object environments, the algorithms presented here can easily be applied in a more general setting. For example, to continuously compute an ε-join[1] between a set of relatively fixed points and another set of points that may move arbitrarily. Such continuous queries over ever-changing data are also important for many applications that handle streaming data from sensors and wireless location-based services in Location-commerce (L-commerce) [6][17]. Towards this goal of wider applicability of our algorithms, we make no assumptions about the speed and nature of the movement of objects or the fraction of objects that can move at any time instant. That is, at any moment in time *all* objects can move with arbitrary speeds in arbitrary directions.

Current efforts at evaluating queries over moving objects have focused on the development of disk-based indexes. The problem of scalable, real-time execution of continuous queries may not be well suited for disk-based indexing for the following reasons: (i) the need to update the index as objects move; (ii) the need to re-evaluate all queries when any object moves; and (iii) achieving very short execution times for large numbers of moving objects and queries. These factors, combined with the drastically dropping main memory costs make main memory evaluation highly attractive. The growing importance of main memory based algorithms has been underscored by the Asilomar report [7], which projects that within 10 years main memory sizes will be in the range of terabytes.

The location of a moving object can be represented in memory as a 2-dimensional point while its other attributes can be stored on disk. One local server is likely to be responsible for handling a limited number of moving objects (e.g. 1,000,000). For such settings, for even large problem sizes, all the necessary data and auxiliary structures, can be easily kept in the main memory of a high-end workstation.

In order for the solution to be effective it is necessary to efficiently compute the matching between large numbers of objects and queries. While multidimensional indexes tailored for main memory, as proposed in [15], would perform better than disk-oriented structures, the use of an index on the moving objects suffers from the need for constant updating as the objects move – resulting in degraded performance. To avoid this need for constant updating of the index structure and to improve the processing of continuous queries, we propose a very different approach: *Query Indexing*. In contrast to the traditional approach of building an index on the moving objects, we propose to build an index on the queries. This approach is especially suited for evaluating continuous queries over moving objects, since the queries remain active for long periods of time, and objects are constantly moving.

In this paper we investigate several in-memory index structures for efficient and scalable processing of continuous queries. We evaluate not only indexes designed to be used in main memory, but also disk-based indexes adapted and optimized for main memory. Our results show that using a grid-like structure gives the best performance, even when the data is skewed. We also propose an effective technique for improving the caching performance. The proposed solutions are extremely efficient, e.g. 100,000 (25,000) continuous queries over 100,000 (1,000,000) objects can be evaluated in as little as 0.288 (0.762) seconds. The use of query indexing is critical for achieving such efficient processing. We also present an analytical evaluation for the optimal grid size. The analysis matches well with the experimental results. A technique for improving the cache hit-rate is developed that achieves a speed up of 100%.

The remainder of this paper is organized as follows. In Section 2 we present related work. Section 3 describes the problem of continuous query processing, Query Indexing, and the index structures considered. We also present an effective technique for improving the cache hit-rate. Section 4 presents the experimental results and Section 5 concludes the paper.

[1]An ε-join between two sets of points is defined as all pairs of points, one from each set, such that the distance between the points is less then ε.

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. Optimization of disk-based index structures has been explored recently for B+-trees [23] and multidimensional indexes [15]. Both studies investigate the redesign of the nodes in order to improve cache performance. Neither study addresses the problem of executing continuous queries or the constant movement of objects (changes to data). The goal of our algorithm is to efficiently and continuously re-generate the mapping between moving objects and queries. Our algorithm makes no assumptions about the future positions of objects. It is also not necessary for objects to move according to well-behaved patterns as in [24]. The problem of scalable, efficient computation of continuous range queries over moving objects is ideally suited for main memory evaluation. To the best of our knowledge no existing work addresses the main memory execution of multiple concurrent queries on moving objects as proposed in the following sections.

Indexing techniques for moving objects have been proposed in the literature, e.g., [3], [18] index the histories, or trajectories, of the positions of moving objects, while [24] indexes the current and anticipated future positions of the moving objects. In [16], trajectories are mapped to points in a higher-dimensional space which are then indexed. In [24], 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 [4, 5, 19, 29], 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. [27] use quad-trees to index the trajectories of one-dimensional moving points. Kollios [16] et al. map moving objects and their velocities into points and store the points in a kD-tree. Pfoser et al. [20][21] 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 [2] through the use of indexing schemes using external range trees. [28] and [30] 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. Issues relating to location dependent database querying are addressed in [25]. Broadcast of data becomes an important technique for scalable communication in the mobile environment. Efficient broadcast techniques are proposed in [1][10][11][12]. Spatio-temporal database models to support moving objects, spatio-temporal types and supporting operations have been developed in [8][9]. A excellent review of multidimensional index structures including grid-like and Quad-tree based structures can be found in [27].

3 Continuous Query Evaluation

3.1 Model of object movement

The issue of obtaining the updated locations of objects is independent of the algorithm used for evaluating the queries. Since the focus of this research is on the efficient evaluation of queries, we assume that updated location information is available at the server, without considering how exactly it is made available. Below we briefly discuss the common assumptions made in order to reduce communication.

The most common assumption is that each object moves on a straight line path with a constant speed, and updates the server with its direction of movement and speed when they change. A similar assumption is that objects are moving with constant speed on a known road. A mutual feature of these assumptions is that for each moving object the server can determine its location based upon a formula. In our experiments new locations of objects are generated at the beginning of each cycle. While some index structures for moving objects rely upon restricted models of movement (e.g. [26]), *Query Indexing* allows objects to move arbitrarily. Therefore, the objects can move anywhere in the domain, but the overall object distribution chosen for the experiment is maintained (uniform, skewed, etc.).

3.2 Query indexing

The problem of continuous query evaluation is: *Given a set of queries and a set of moving objects, continuously determine the set of objects that are contained within each query.* Notice that this approach can be easily extended to compute object to query mapping, handle region queries, answer simple density queries (e.g. monitor how many people are in a building), compute similarity join for multidimensional data (e.g., [13]).

Clearly, with a large number of queries and moving objects, it is infeasible to re-evaluate each query whenever any object moves. A more practical approach is to re-evaluate all queries periodically taking into account the latest positions of all objects. In order for the results to be useful, a short re-evaluation period is desired. The goal of the query evaluation algorithms is therefore to re-evaluate all queries in as short a time as possible.

A naïve solution is to compare each object against each query in every period. Another approach is building an index (e.g. R-tree) on the objects to speed up the queries. Although for regular queries this would result in improvements in performance over the naïve approach, it suffers a major drawback for the moving objects environment: the index needs to be continuously updated as object positions change. Maintaining such an index on mobile data is a challenging task [24].

This paper builds on the idea of *Query Indexing* proposed in [22]. Instead of building an index on the moving objects (which would require frequent updating), create an index on the more stable queries. Any spatial index structure can be used to build the query index (e.g. R*-tree, Quad-tree, etc).

```
Input: Datasets $X$, $Q$, and index $I_Q$ on $Q$
Output: All $q_j.S_X$, for $j = 1, \ldots, |Q|$
1. for $i \leftarrow 1$ to $|Q|$ do
        (a) $q_i.S_X \leftarrow \emptyset$
2. for $i \leftarrow 1$ to $|X|$ do
        (a) Get $x_i.S_Q$ using index $I_Q$
        (b) for $j \leftarrow 1$ to $|x_i.S_Q|$ do
                i. $q \leftarrow x_i.S_Q[j]$
                ii. $q.S_X \leftarrow q.S_X \cup \{x_i\}$
```

Figure 1: Query Indexing approach: a cycle processing

The evaluation of continuous queries in each cycle proceeds as follows. Let $X = \{x_1, \ldots, x_{|X|}\}$ be the set of moving objects. Let $Q = \{q_1, \ldots, q_{|Q|}\}$ be the set of queries. Let $x_i.S_Q$ be the set of all queries in which object x_i is contained, $x_i.S_Q = \{q : x_i \in q;\ q \in Q\}$. Let $q_j.S_X$ be the set of all objects contained in query q_j, $q_j.S_X = \{x : x \in q_j;\ x \in X\}$. The goal is to compute all $q_j.S_X$, for $j = 1, \ldots, |Q|$, based upon the current locations of the objects by the end of each cycle. Since at any time instant all objects move with arbitrary speeds and directions, all $x_i.S_Q$'s and $q_j.S_X$'s are likely to be completely different from one cycle to the next. Consequently, incremental solutions are of little value. In each cycle, we first use the query index to compute $x_i.S_Q$ for each object x_i, as shown in Figure 1. Next, for each query q in $x_i.S_Q$, we add x_i to $q.S_X$.

Some important consequences of indexing the queries instead of the data should be noted. Firstly, the index needs no modification unless there is a change to the set of queries – a relatively less frequent event in comparison to changes to object locations since we are dealing with continuous queries. Secondly, the location of each object can change greatly from one cycle to the next without having any impact on the performance. In other words, there is no restriction on the nature of movement or speed of the objects. There is also no restriction on the fraction of object that can move at any moment in time, that is we assume that all objects move. This is an advantage since many known object indexing techniques rely upon certain assumptions about the movement of objects.

Next we discuss the feasibility of in-memory query indexing by evaluating different types of indexes for queries. Clearly, if we are unable to select an appropriate index, the time needed to complete one cycle can be large (e.g. 1 min) and the approach would be unacceptable. Below we briefly discuss indexing techniques for building a query index in main memory. In Section 4 we evaluate the performance of these alternative indexes.

3.3 Indexing techniques

We consider the following five well-known index structures: R*-tree, R-tree, CR-tree, quad-tree, and grid. The R-tree and R*-tree index structures are designed to be disk-based structures. The CR-tree [15], on the other hand, is a variant of R-trees optimized for main memory. All of these indexes were implemented for main memory use. In order to make a fair comparison, we did not choose large node sizes for these trees. Instead, we experimentally determined the best choice of node size for main-memory evaluation. All three indexes (R*-tree, R-tree, and CR-tree) showed best performance when the number of entries per node was chosen to be five. This value was used for all experiments. Details of the CR-tree are described in [15]. The main idea is to make R-tree cache-conscious by compressing MBRs. This is achieved by using so-called Quantized Relative Minimum Bounding Rectangles (QRMBR). Other, well-known prior to that publication, optimizations have also been used in the paper. We implemented the CR-tree based upon the main idea of QRMBRs without the other optimizations.

Because many variations exist, we describe the grid index as it is used here for query indexing. The grid index is a 2-dimensional array of “cells”. Each cell represents a region of space generated by partitioning the domain using a uniform grid. Figure 2 shows an example of a grid. Throughout the paper, we assume that the domain is normalized to the unit square.

In this example, the domain is divided into a 10×10 grid of 100 cells, each of size 0.1×0.1. Since we have a uniform grid, given the coordinates of an object, it is easy to calculate the cell that it falls under in O(1) time. Each cell contains two lists that are identified as *full* and *part* (see Figure 2). The *full* (*part*) list of a cell contains *pointers* to all the queries that fully (partially) cover the cell. The choice of data structures for the *full* and *part* lists is critical for performance. We implemented these lists as *dynamic arrays*[2] rather than *lists*: this improves performance by roughly 40% due to the achieved clustering. An analytical solution for the appropriate choice of grid size

[2]A dynamic array is a standard data structure for arrays whose size adjusts dynamically.

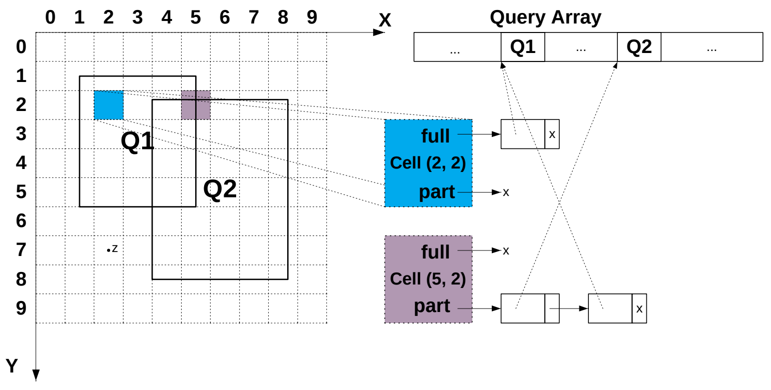


Figure 2: Example of grid

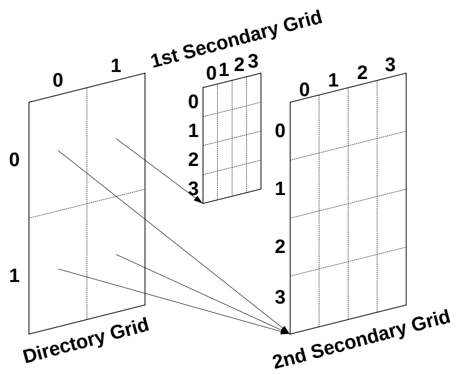


Figure 3: Example of grid with two tiers

is presented in Section 3.4. As will be seen in the experimental section, this simple one-level grid index outperforms all other structures for uniform as well as skewed data. However, for the case of highly skewed data (e.g. roughly half the queries fall within one cell), the *full* and *part* lists grow too large. Such situations are easily handled by switching to a two-tier grid. If any of the lists grows beyond a threshold value, the grid index converts to a directory grid and a few secondary grids, see Figure 3. The directory grid is used to determine which secondary grid to use. Each directory grid cell points to a secondary grid. The secondary grid is used in the same way as the one-level grid. While this idea of generating an extra layer can be applied as many times as is necessitated by the data distribution, three or more layers are unlikely to lead to better performance in practice. Consider for example, that the domain of interest represents a 1000-kilometer by 1000-kilometer region. With a 1000×1000 grid, a cell of the two-tier grid corresponds a square of side 1 meter – it is very unlikely that there are very many objects or queries in such a small region in practice.

Observe that the grid index and the quad-tree are closely related. Both are space partitioning and split a region if it is overfull. There is, however, a subtle difference: the grid index avoids many conditional ("if") branches in its search algorithm due to its shorter known height. Furthermore, the grid index is not a tree since siblings can point to a common "child" [14], see Figure 3.

Another advantage of the grid is that it typically has far more cells per level. A quad-tree therefore can be very deep, especially for skewed data. We expect that a quad-tree like structure that has more cells per level would perform better than a standard quad-tree. In order to test this hypothesis, we also consider what we call a 32-tree. The 32-tree is identical to a quad-tree, except that it divides a cell using a 32×32 grid, compared to the 2×2 grid used by the quad-tree. Pointers to children are used instead of keeping an array of pointers to children. In order to further improve performance, we implemented the following optimization: In addition to leaf nodes, internal nodes can also have an associated *full* list. Only leaf nodes have a *part* list. A *full* list contains all queries that fully cover the bounding rectangle (BR) of the node, but do not fully cover the BR of its parent. Adding a rectangle (or region) to a node proceed as follows. If the rectangle fully covers the BR, it is added to the *full* list and the algorithm stops for that node. If this is a leaf node and there is space in the *part* list, the rectangle is added to *part* list. Otherwise the set of all relevant children is determined and the procedure is applied to each of them.

Storing *full* lists in non-leaf nodes has two advantages. One advantage is saving of space: without such lists, when a query fully covers a node's BR it would be duplicated in all the node's children. A second advantage is that it has the potential to speed up point queries. If a point query falls within the BR of a node then it is relevant to all queries in the *full* list of this node – no further checks for these queries are needed. A leaf node split is based on the *part* list size only. While many more optimizations are possible, we did not explore these further. The purpose of studying the 32-tree is to establish the generality and flexibility of the grid-based approach.

3.4 Choosing grid size

We now present an analysis of appropriate choice for the cell size for the grid index in the context of main-memory query indexing. Consider the case where m square queries with side q, uniformly

Table 1: Parameters for choosing grid size

Param	**Meaning**
$\mathcal{D}$	Domain, $\mathcal{D} = [0,1]^2$
m	The number of queries
q	The length of query side
c	The length of cell side
i	$i = \lfloor \frac{q}{c} \rfloor$
x	$x = q - i \times c$

distributed on $[0,1]^2$ domain, are added to index, see Figure 4. Let c be the side of each cell. Then query side q can be presented as

$$q = i \times c + x;\ i = \left\lfloor \frac{q}{c} \right\rfloor,\ x = q - i \times c.$$

Cell $G(0,0)$ can be logically divided into three parts (or sets), see Figure 4. *Set0* is the top-left rectangle of size $(c-x)^2$, *Set1* is the bottom-left and top-right rectangles of combined size $2x(c-x)$, and *Set2* is the bottom-right rectangle of size x^2. We now analyze the average number of cells partly covered by a query. Without loss of generality let us consider the case where the top-left corner of query Q is somewhere within cell $G(0,0)$.

Case 1: Query side is greater than cell side ($q > c$). It can be verified that if the top-left corner of Q is inside *Set0*, then Q is present in $4i$ part lists (see Figure 5), for *Set1* this number is $4i + 2$ (see Figure 6), and for *Set2* it is $4i + 4$ (see Figure 7). Assuming the uniform distribution, the probability that the top-left corner of Q is inside *Set0* is $\frac{(c-x)^2}{c^2}$, inside *Set1* is $\frac{2x(c-x)}{c^2}$, and inside

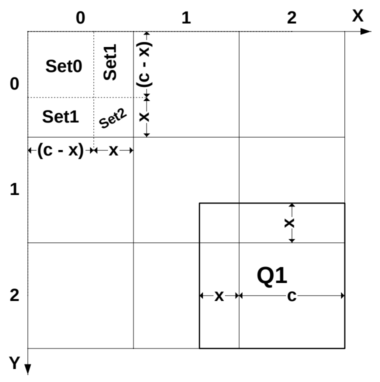


Figure 4: Choosing grid size

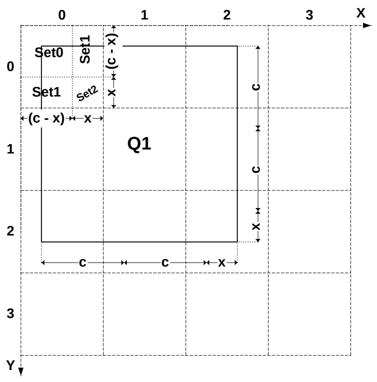


Figure 5: Top-left corner in *Set0*

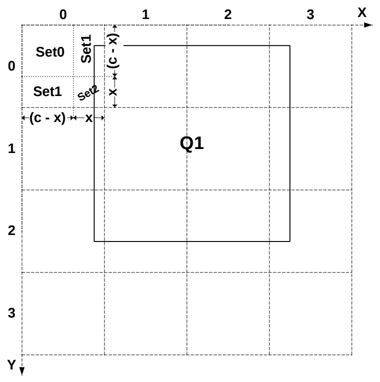


Figure 6: Top-left corner in *Set1*

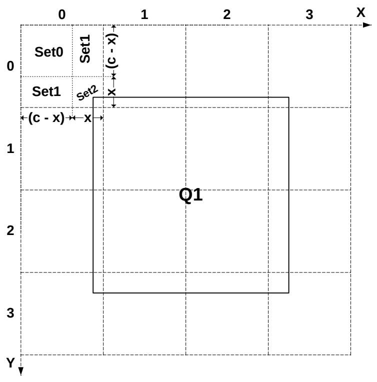


Figure 7: Top-left corner in *Set2*

Set2 is $\frac{x^2}{c^2}$. Therefore, on the average, each query ends up in avg number of part lists:

$$\begin{aligned} avg &= \frac{[4i(c-x)^2 + (4i+2)2x(c-x) + (4i+4)x^2]}{c^2} \\ &= \frac{4c[x+ic]}{c^2} \\ &= \frac{4q}{c} \end{aligned}$$

Correspondingly, m queries end up in $\frac{4qm}{c}$ part lists. Since the total number of cells is $\frac{1}{c^2}$, each cell has part list of size $4qmc$ on average.

In our implementation of the algorithm the difference in time needed to process a cell when its part list is empty vs. when its part list has size one is very small. When choosing cell size c such that $4qmc = 1$, that is $c = \frac{1}{4qm}$, on average, the size of a cell's part list is one and choosing smaller cell size is unnecessary. Since we consider the case where query size is greater than cell size ($q > c$), the following must also be true: $q > \frac{1}{4qm}$ and therefore $q > \frac{1}{2\sqrt{m}}$.

Case 2: Query side is less than cell side ($q < c$). In this case $i = \lfloor \frac{q}{c} \rfloor = 0$ and $x = q - i \times c = q$. It can be verified that if the top-left corner of Q is inside *Set0*, then Q is present in 1 part list, for *Set1* this number is 2, and for *Set2* it is 4, see Figures 5, 6, and 7. The probabilities that the top-left corner is inside *Set0*, *Set1*, or *Set2* remain the same. Therefore, the average number of part lists, each query ends up in is computed as:

$$\begin{aligned} avg &= \frac{[1 \times (c-x)^2 + 2 \times 2x(c-x) + 4 \times x^2]}{c^2} \\ &= \frac{(x+c)^2}{c^2} \end{aligned}$$

Correspondingly, m queries end up in $\frac{m(x+c)^2}{c^2}$ part lists. Since the total number of cells is $\frac{1}{c^2}$, each cell has part list of size $m(x+c)^2$ on average.

In our implementation of the algorithm the difference in time needed to process a cell when its part list is empty vs. when its part list has size one is very small. By choosing cell size c such that $m(x+c)^2 = 1$, that is $c = \frac{1}{\sqrt{m}} - x$, on average, the size of each cell part list is one and choosing smaller cell size is unnecessary. Since we consider the case where query size is less than cell size ($q < c$), and since $q = x$ for this case, the following must also be true $q < \frac{1}{\sqrt{m}} - q$ and therefore $q < \frac{1}{2\sqrt{m}}$.

Final formula The final formula for choosing cell size c is:

$$c = \begin{cases} \frac{1}{4qm} & \text{if } q > \frac{1}{2\sqrt{m}}; \\ \frac{1}{\sqrt{m}} - q & \text{otherwise.} \end{cases}$$

Smaller values of cell size c give only minor performance improvement while incurring large memory penalty. Remember, the grid size is computed as $1/c$, and as the result we have $\frac{1}{c} \times \frac{1}{c}$ cell grid.

Let us consider the example in Figure 13a. There the number of queries m is $25,000$ and the query size q is 0.01. We first need to test the condition $q > \frac{1}{2\sqrt{m}}$. Since $\frac{1}{2\sqrt{m}} = 0.001$ is less than $q = 0.01$, then $c = \frac{1}{4qm}$ formula should be used, and therefore cell size $c = 0.001$. This means that grid should be of size 1000×1000 cells (i.e., $\frac{1}{c} \times \frac{1}{c}$) and a finer grid gives only minor performance improvement while incurring a large memory penalty. We study the impact of tile size in the experimental section and show that the results match the analytical prediction.

3.5 Improving cache hit-rate

The performance of main-memory algorithms is greatly affected by cache hit-rates. In this section we describe an optimization that can drastically improve cache hit-rates (and consequently the overall performance) for the query indexing approach. In each cycle the processing involves searching the index structure for each object's current location in order to determine the queries that cover the objects current location.

For each object, its cell is computed, and the *full* and *part* lists of this cell are accessed. The algorithm simply processes objects in sequential order in the array. Consider the example shown in Figure 8. The order in which the objects appear in the array is shown on the left of the figure in the "Unsorted Point Array". Note that we use the terms object and point interchangeably. In this example cell $G(0,0)$ and its lists are accessed for processing object P_2 and then later for processing object P_n. Since several other objects are processed after P_2 and before P_n, it is likely that cell $G(0,0)$ and its lists are not in the cache when P_n is processed – resulting in cache misses.

If objects are to maintain their locality, then the cache hit-rate can be improved by altering the order of processing the objects. Objects are reordered in the array such that objects that are close together in our $[0,1]^2$ 2-dimensional domain are also tend to be close together in the object array, as in the array on the right labeled "Sorted Point Array" shown in Figure 8. With this ordering, object P_2 is analyzed first and therefore cell $G(0,0)$ and its lists are processed. Then object P_n is

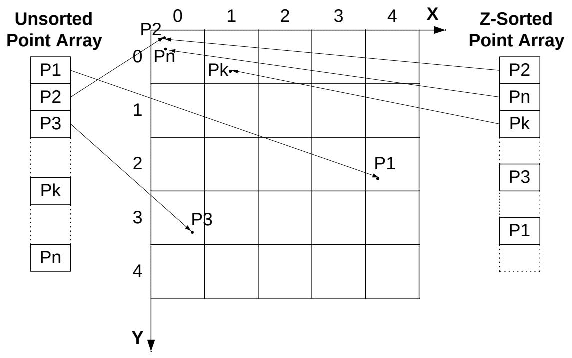


Figure 8: Example of un-sorted and z-sorted object arrays

analyzed and cell $G(0,0)$ and its lists are processed again. In this situation everything relevant to cell $G(0,0)$ is likely to remain in the CPU cache after the first processing and is reused from the cache during the second processing. The speed up effect is also achieved because objects that are close together are more likely to be covered by the same queries than objects that are far apart, thus queries are more likely to be retrieved from the cache rather than from main memory.

Sorting the objects to ensure that objects that are close to each other are also tend to be close in the array order can easily be achieved. One possible approach is to group objects by cells – i.e. all objects that fall under each cell are placed adjacently in the array and thus processed together. The objects grouped by cell can then be placed in the array using a row-major or column-major ordering for the cells. Although this is effective, the benefit of the sorting is lost if the object moves out of its current cell and enters an adjacent cell that is not close by in the ordering used for the cells (e.g object moves to adjacent cell in next row and row major ordering is used). We propose an alternative approach: order the points using any of the well-known space filling curves such as Z-order or Hilbert curve. We choose to use a sorting based on the Z-order. Z-sorting significantly improves the performance of the main memory algorithm, as will be shown in the experiment section.

It is important to understand that *the use of this technique does not require that objects have to preserve their locality.* The only effect of sorting the objects according to their earlier positions is to alter the order in which objects are processed in each cycle. The objects are still free to move arbitrarily. Of course, the *effectiveness* of this technique relies upon objects maintaining their locality over a period of time. If it turns out that objects do not maintain their locality then we are, on the average, no worse than the situation in which we do not sort. Thus, for the case where objects preserve locality sorting the objects based upon their location at some time can be beneficial. It should also be noted that the exact position used for each object is not important. Thus the sorting can be carried out infrequently, say once a day.

4 Experimental Results

In this section we present the performance results for the index structures. The results report the actual times for the execution of the various algorithms. First we describe the parameters of the experiments, followed by the results and discussion.

In all our experiments we used a 1GHz Pentium III machine with 2GB of memory. The machine has 32K of level-1 cache (16K for instructions and 16K for data) and 256K level-2 cache. Moving objects were represented as points distributed on the unit square $[0,1] \times [0,1]$. The number of objects ranges from 100,000 to 1,000,000. Range-queries were represented as squares with sides 0.01. Experiments with other sizes of queries yielded similar results and thus are omitted. For distributions of objects and queries in the domain we considered the following cases:

1. **Uniform**: Objects and queries are uniformly distributed.

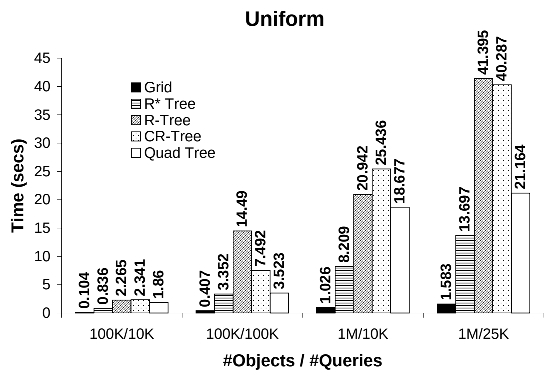
2. **Skewed**: The objects and queries are distributed among five clusters. Within each cluster objects and queries are distributed normally with a standard deviation of 0.05 for objects and 0.1 for queries.

3. **Hyper-skewed**: Half of the objects (queries) are distributed uniformly on $[0, 1] \times [0, 1]$, the other half on $[0, 0.001] \times [0, 0.001]$. Queries in $[0, 0.001] \times [0, 0.001]$ are squares with sides 0.00001 to avoid excessive selectivity.

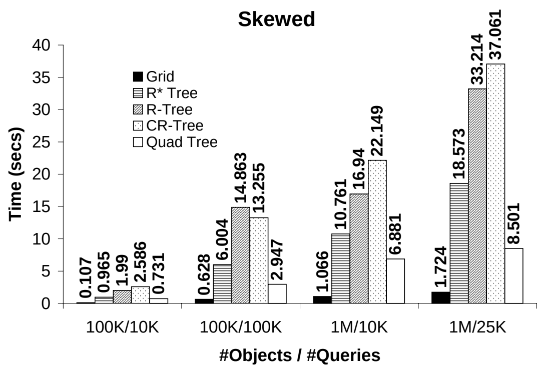
We consider the skewed case to be most representative. The hyper-skewed case represents a pathological situation designed to study the performance of the schemes under extreme skew. In the majority of our experiments the grid was chosen to consist of 1000×1000 cells. The testing proceeds as follows: first, queries and objects are generated and put into arrays. Then the index is initialized and the queries are added to it. In each cycle first the locations are updated then the query results are evaluated.

4.1 Comparing efficiency of indexes

Figure 9a shows the results for various combinations of number of objects and queries with uniform distribution. The y-axis gives the processing time for one cycle in seconds for each experiment. Figure 9b shows similar results for the skewed case.



(a)



(b)

Figure 9: Index comparison for (a) Uniform distribution (b) Skewed distribution

Each cycle consists of two steps: (i) *moving* objects (i.e., determining current object locations) and (ii) *processing/evaluation*. From Figure 9b we can see that for the case of 100,000 objects and 100,000 queries the evaluation step for the grid takes 0.628 seconds. Updating (determining) object locations takes 0.15 seconds for 10^5 objects and 1.5 seconds for 10^6 objects on average. Thus the length of each cycle is just 0.778 seconds on average.

The grid index gives the best performance in all these cases. While the superior performance of the grid for the uniform case is expected, the case for skewed data is surprising. For all experiments the grid index consisted of only a single level.

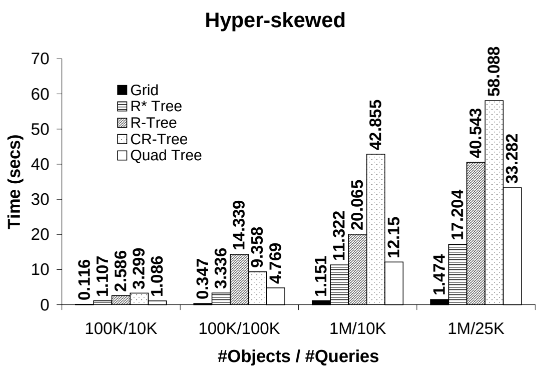
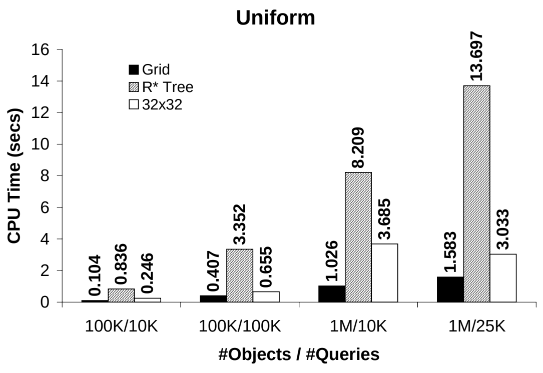


Figure 10: Index comparison for Hyper-skewed distribution

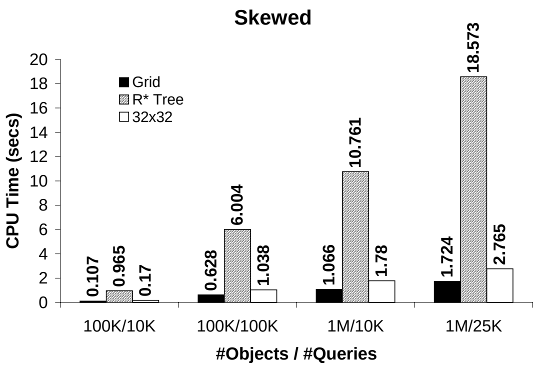
Figure 10 shows the results for the hyper-skewed case. For the hyper-skewed case, the grid switches from one-tier to two-tier due to an overfull cell. The processing time for a simple one-tier grid is too high, as expected, and is not shown on the figure. It is interesting to see that the grid index once again outperforms the other schemes. There is a significant difference in performance of grid and the other approaches for all three distributions. For example, with 1,000,000 objects and 25,000 queries, grid evaluates all queries in 1.724 seconds as compared to 33.2 seconds for the R-Tree, and 8.5 seconds for the Quad Tree. This extremely fast evaluation implies that with the grid index, the cycle time is very small – in other words, we can re-compute the set of objects contained in each query every 3.2 seconds or faster (1.7 seconds for the evaluation step plus 1.5 seconds for updating the locations of objects). This establishes the feasibility of in-memory query indexing for managing continuous queries.

4.2 32-tree index

It can be seen that the quad-tree performs better than R-tree like data structures for skewed cases, but worse for the majority of the uniform and hyper-skewed cases.



(a)



(b)

Figure 11: Performance of 32-tree (a) Uniform distribution (b) Skewed distribution

The problem with the quad-tree for hyper-skewed case is that it has a large height. This suggests that if it were able to "zoom" faster it would be a better index than R*-tree. We test

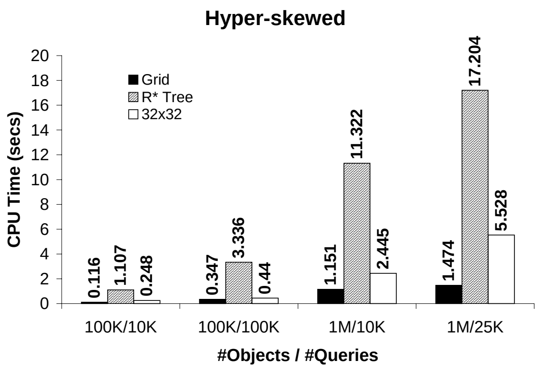


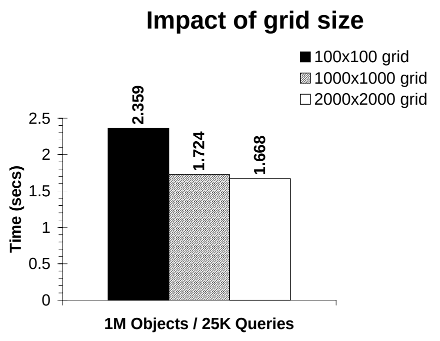
Figure 12: Performance of 32-tree for Hyper-skewed distribution

this hypothesis by evaluating the 32-tree which is similar to the quad-tree except that it has more divisions at each node.

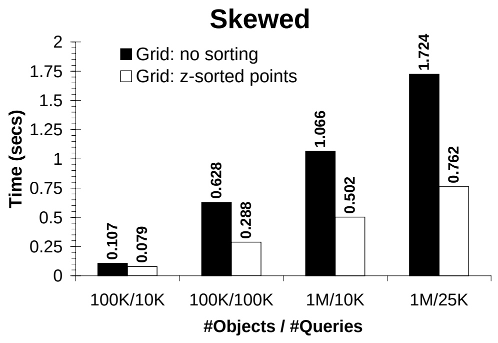
The performance of the 32-tree along with that for the grid and R*-tree for uniform, skewed, and hyper-skewed data is shown in Figures 11 and 12. As can be seen from the figures our hypothesis is true: the performance of the 32-tree lies between that of the R*-tree and the grid.

4.3 Choice of grid size

In this experiment we study the impact of the number of cells in the grid. The analysis in Section 3.4 predicted that a 1000×1000 grid should be chosen for the settings of this example (uniform, 1M objects, 25K queries). Figure 13a presents the processing time needed with grid sizes 100×100,



(a)



(b)

Figure 13: a) Impact of grid size on processing time. b) Effectiveness of Z-Sorting.

1000×1000, and 2000×2000 cells. As can be seen, increasing the number of cells has the effect of reducing the average number of queries for a cell thereby reducing the processing time. There is a substantial increase in performance as we move from 100×100 cells to 1000×1000 cells. The increase is minor when we move from 1000×1000 to 2000×2000 cells for our case of 1M objects and 25K queries. This behavior validates the analytical results.

4.4 Z-sort optimization

Figure 13b illustrates the effect of the Z-sort technique on evaluation time for ideally Z-sorted data. Z-sorting reorders the data such that objects that are close together tend to be processed close together. When processing each object in the array from the beginning to the end, objects that are close to each other tend to reuse information stored in the cache rather than retrieve it from main-memory. From the results, we see that sorting objects improves the performance by roughly 50%. The Z-sort technique was used only in this experiment.

4.5 Grid: performance of adding and removing queries

We now study the efficiency of modifying the grid index. The results in Figure 14 show how long it takes to add and remove queries to/from an existing index that already contains some queries. Although modifications to queries are expected to be rare, we see that adding or removing queries

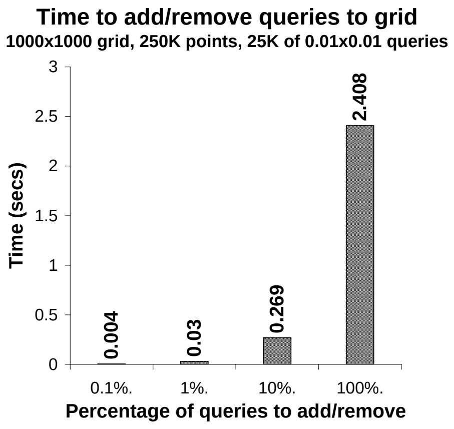


Figure 14: Adding and deleting queries.

is done very efficiently with the grid. For example, the 100% bar shows that 100% of 25K queries can be added or deleted in only 2.408 seconds. The decision whether to add or delete a query at a particular step is made with probability of 0.5 for each query. Therefore we see that even significant changes to the query set can be effectively handled by the grid approach.

5 Summary

In this paper we presented a Query-Index approach for in-memory evaluation of continuous range queries on moving objects. We established that the proposed approach is in fact a very efficient solution even if there are no limits on object speed or nature of movement or fraction of objects that move at any moment in time, which are common restrictions made in similar research. We presented results for seven different in-memory spatial indexes. The grid approach showed the best result even for the skewed case. A technique of sorting the objects to improve the cache hit-rate was presented. The performance of the grid index was roughly doubled with this optimization. An analysis for selecting optimal grid size and experimental validation was presented. We also showed that even though the set of continuous queries is to remain almost unchanged, nevertheless grid can very efficiently add or remove large numbers of queries. Overall, indexing the queries using the grid index gives orders of magnitude better performance than other index structures such as R*-trees.

References

[1] 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.

[2] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, May 15-17 2000.

[3] 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.

[4] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Querying imprecise data in moving object environments. *TKDE, IEEE Transactions on Knowledge and Data Engineering.* conditionally accepted.

[5] Reynold Cheng, Sunil Prabhakar, and Dmitri V. Kalashnikov. Querying imprecise data in moving object environments. In *ICDE'03, Proc. of the 19th IEEE International Conference on Data Engineering*, Bangalore, India, March 5–8 2003.

[6] US Wireless Corp. The market potential of the wireless location industry. http://www.uswcorp.com/USWCMainPages/laby.htm.

[7] P. Bernstein et. al. The asilomar report on database research. *SIGMOD Record*, 27(4):74–80, 1998.

[8] 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.

[9] 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.

[10] S. E. Hambrusch, C.-M. Liu, W. Aref, and S. Prabhakar. Minimizing broadcast costs under edge reductions in tree networks. In *7th International Symposium on Spatial and Temporal Databases (SSTD 2001)*, July 2001.

[11] 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.

[12] 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.

[13] Dmitri V. Kalashnikov and Sunil Prabhakar. Similarity join for low- and high- dimensional data. In *DASFAA'03, Proc. of the 8th International Conference on Database Systems for Advanced Applications, IEEE Computer Society Press*, Kyoto, Japan, March 26–28 2003.

[14] Dmitri V. Kalashnikov, Sunil Prabhakar, Walid Aref, and Susanne Hambrusch. Efficient evaluation of continuous range queries on moving objects. Technical Report TR 02-015, Department of Computer Science, Purdue University, West Lafayette, IN 47907, June 2002.

[15] K. Kim, S.K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *SIGMOD'01, Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Santa Barbara, CA, USA, May 21–24 2001.

[16] 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.

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

[18] Anil Kumar, Vassilis J. Tsotras, and Christos Faloutsos. Designing access methods for bitemporal databases. 10(1):1–20, 1998.

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

[20] 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.

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

[22] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, October 2002.

[23] Jun Rao and Kenneth A. Ross. Making B^+-trees cache conscious in main memory. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 475–486. ACM, 2000.

[24] 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.

[25] Ayse Y. Seydim, Margaret H. Dunham, and Vijay Kumar. Location dependent query processing. In *MobiDE01, Second ACM International Workshop on Data Engineering for Mobile and Wireless Access*, Santa Barbara, California, USA, May 2001.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.