2002

In-Memory Evaluation of Continuous Range Queries on Moving Objects

Dmitri V. Kalashnikov

Sunil Prabhakar
Purdue University, sunil@cs.purdue.edu

Report Number:
02-003

IN-MEMORY EVALUATIN OF CONTINUOUS RANGE QUERIES ON MOVING OBJECTS

Dmitri V. Kalashnikov
Sunil Prabhakar

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #02-003
January 2002

In-Memory Evaluation of Continuous Range Queries on Moving Objects

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

Abstract

In this paper we address the emerging problem of evaluating continuous range queries over collections of moving objects. Such queries form the basis for a large class of interesting queries e.g., monitoring a region of space to determine which objects (people, vehicles, etc.) enter or leave, and ensuring that the number of policemen in a stadium does not drop below some threshold during a ball game. We present an elegant in-memory algorithm for very fast computation of solutions to the problem of computing both the object-to-query and query-to-object mappings continuously as the objects move. Our techniques are able to efficiently handle changes to the set of queries as well continuously evolving queries such as would be encountered when the query region follows a path through space.

In order for the solution to be effective it is necessary to repeatedly compute the mappings between large numbers of objects and queries in a very short amount of time. To achieve these goals our algorithm uses a novel in-memory hash-like data structure, which we call the CELL-Index. The CELL-Index is very effective, e.g. given a set of 250K uniformly distributed objects and 25K uniformly distributed continuous range queries the algorithm is able to compute point-to-query mapping in 0.347 sec and both mappings in 0.749 sec on a PIII 1GHz machine with 2GB of memory. To further improve performance, we propose the use of a space-filling curve to order the objects. We use the z-ordering to ensure that objects that are close to each other in 2D space are processed together. We call this the Z-Sort optimization. With Z-Sort the above execution times are reduced to 0.24 sec and 0.437 sec respectively. For larger datasets we show that Z-Sort can improve performance by as much as a factor of 2 or 3. The algorithm is able to compute point-to-query mapping for as many as 1,000,000 objects and 100,000 queries, both with skewed distributions.

Object indexing techniques that require continuous updates to the index as objects move cannot yield these levels of performance. We present extensive experimental results that demonstrate that the CELL-Index and Z-Sort techniques are extremely fast and scalable for continuously evaluating range queries over moving objects.

1 Introduction

In this paper we address the emerging problem of evaluating continuous range queries over collections of moving objects. Such queries form the basis for a large class of interesting queries e.g., monitoring a region of space to determine which objects (people, vehicles, etc.) enter or leave, or ensuring that the number of policemen in a stadium does not drop below some threshold during a ball game. Such environments are characterized by a set of queries that are repeatedly evaluated over a large set of data

points that are constantly changing (e.g. moving objects). While the focus of our work is on moving object environments, the techniques presented here can easily be applied in a more general setting. For example, to continuously compute an ε-join 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 L-commerce [7][16][18][19][30]. Towards this goal of wider applicability of our techniques, we make no assumptions about the nature of the movement of objects. Furthermore, while applications may generally require a point-to-query mapping only, we generate both point-to-query and query-to-point mappings. Depending upon the application, different subsequent processing can be applied to the resulting mapping.

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 is not well suited for disk-based indexing for the following reasons: (i) the need to update the index as objects move; (ii) 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 (1GB as one DIMM currently costs $164 -- it cost $200 just a couple of months ago) makes main memory evaluation highly attractive. The growing importance of main memory based algorithms has been underscored by the Asilomar report [6] which projects that within 10 years main memory sizes will be in the range of terabytes. Recent research efforts have also been focusing on main memory issues. For example, the reorganization of disk-based index structures such as B+-Trees and multi-dimensional indexes to improve cache performance in main memory have been developed in [23] and [14] respectively.

In order for the solution to be effective it is necessary to compute the mappings between large numbers of objects and queries in a very short amount of time. While multidimensional indexes tailored for main memory as proposed in [14] 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. We present an elegant in-memory algorithm for very fast computation of solutions to the problem of computing both the object-to-query and query-to-object mappings continuously as the objects move.

Our algorithm uses a novel in-memory hash-like data structure for indexing queries, which we call the CELL-Index. The CELL-Index is very effective, e.g. given a set of 250K uniformly distributed objects

and 25K uniformly distributed continuous range queries the algorithm is able to compute the point-to-query mapping in 0.347 sec and both mappings in 0.749 sec on a PIII 1GHz machine with 2GB of memory. The maximum amount of memory used for the data structures was 391MB (with highly skewed data and queries). To further improve performance, we propose the use of a space-filling curve to order the objects. This ordering of objects achieves higher cache hit rates. We call this the Z-Sort optimization. With Z-Sort the above execution times are reduced to 0.24 sec and 0.437 sec respectively. For larger datasets we show that Z-Sort can improve performance by as much as a factor of 2 or 3. Object indexing techniques that require continuous updates to the index as objects move cannot yield these levels of performance. Our algorithm is able to compute point-to-query mapping for as many as 1,000,000 objects and 100,000 queries, both with skewed distributions.

Continuous queries remain in effect over a period of time, and thus they do not change as rapidly as the movement of the objects over which they are evaluated. We expect that the set of queries does not change very frequently. Certain types of continuous queries may change as rapidly as the objects. For example, navigation queries where the spatial region of the query follows a moving object. Our techniques are able to efficiently handle changes to the set of queries as well continuously evolving navigation queries. We present extensive experimental results that demonstrate that the CELL-Index and Z-Sort techniques are extremely fast and scalable for continuously evaluating range queries over moving objects.

The remainder of this paper is organized as follows. In section 2 we present related work, Section 3 describes the CELL-Index and the evaluation of continuous queries. Section 4 presents the Z-Sort optimization, Section 5 describes how navigation queries can be efficiently handled. Section 6 presents the experimental results 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. Optimization of disk-based index structures has been explored recently for B^+-trees [23] and multidimensional indexes [14]. 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 technique is to efficiently and continuously re-generate the mapping between moving objects and queries. The technique 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 [25]. Due to the characteristics of the problem it is ideally suited for main memory execution. 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 are being proposed in the literature, e.g., [4], [17]index the histories, or trajectories, of the positions of moving objects, while [25] indexes the current and anticipated future positions of the moving objects. In [15], trajectories are mapped to points in a higher-dimensional space which are then indexed. In [25], 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 [20][34], 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. [29] use quadtrees [26] to index the trajectories of one-dimensional moving points. Kollios [15] et. al. map moving objects and their velocities into points and store the points in a kD-tree. Pfoser et. al [21][22] 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. [33] and [35] 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 [27]. Broadcast of data becomes an important technique for scalable communication in the mobile environment. Efficient broadcast techniques are proposed in [1][2][10][11][12][13][36]. Spatio-temporal database models to support moving objects, spatio-temporal types and supporting operations have been developed in [8][9].

3 Continuous Query Evaluation with CELL-Index

In this section we discuss the evaluation of continuous queries using the CELL-Index. The problem is as follows:

Given a set of queries and a set of moving objects, continuously evaluate the set of objects that fall within each query (query-to-point mapping), and the set of queries containing each point (point-to-query mapping).

Our interest is only in producing these mappings, we are not concerned with the details of the use of these mappings. We assume that the evaluation proceeds as follows:

```
while(true)
{
      while (period of time T)
      {
            produce mappings
            consume mappings
            make minor adjustments to index (e.g. navigation queries)
      }
      make major adjustments to index (add/remove many queries)
}
```

The mappings are generated continuously in each iteration of the inner while loop. The loop is executed for a period of time (T). In each iteration, the mappings are generated and consumed, followed by adjustments to the CELL-Index to reflect the changes in queries that evolve rapidly, for example the navigation queries. The mappings are computed using an array containing the latest positions of the objects and the CELL-Index on the queries. Once every T seconds, changes to the set of queries are made, for example the addition or removal of queries. The CELL-Index is used to evaluate the two mappings.

The CELL-Index is a data structure maintained in memory. It consists of a two-dimensional array of "cells". Each cell represents a region of space generated by partitioning the domain using a uniform grid. Figure 3-1 shows an example of a CELL-Index. The domain corresponding to the unit square, [0, 1] x [0, 1] is divided into a 10x10 grid of 100 cells, each of size 0.1 x 0.1. Since we have uniform griding, given the coordinates of an object, it is easy to calculate the cell that it falls under. The cell coordinates for a point `z(x,y)` are easily computed as:

```
cell_x = (int) x / hX;
cell_y = (int) y / hY;
```

where `x` and `y` are coordinates of the point we are interested in, `hX` and `hY` are the horizontal and vertical size of each cell (e.g. in our example hX=hY=0.1). Applying the formula above we can determine the cell that a point belongs to in O(1) time. Each cell contains two lists that are identified as *full* and *part* (see Figure 3-1). Both lists are implemented as linked lists with each entry containing a pointer to a query and a *next* pointer. The *full* list of a cell contains pointers to all the queries that fully cover the cell. The *part* list of each cell contains pointers to all the queries that only partially cover the cell.

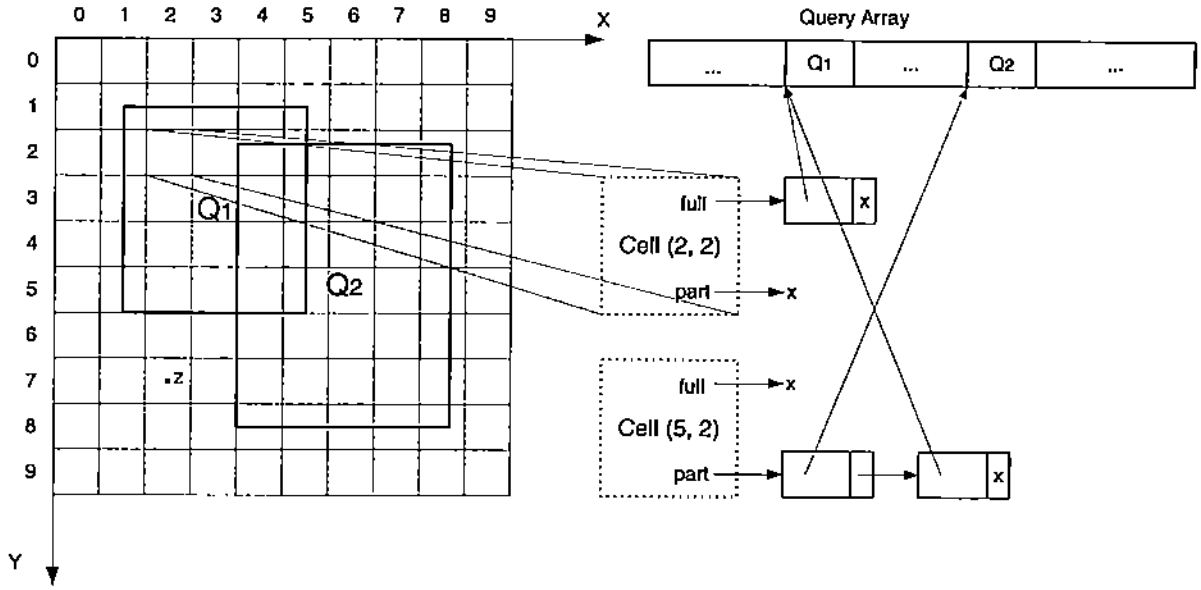

Figure 3-1 Example of CELL-Index

The CELL-Index is constructed by adding queries to it one by one. The addition of a query involves: (i) identifying the cells that the query covers; (ii) identifying the appropriate lists to which the query must be added for each covered cell; and (iii) adding the query to these lists. For example, query Q_1 in Figure 3-1 covers cells [1, 2, 3, 4, 5] x [1, 2, 3, 4, 5]. In cells [1, 2, 3, 4, 5] x [1], [1] x [2, 3, 4, 5], and [5] x [2, 3, 4, 5] pointers to query Q_1 are added to the *part* lists. For the remaining cells (i.e. cells [2, 3, 4] x [2, 3, 4, 5]) pointers to query Q_1 are added to the *full* lists.

In order to find the point-to-query mapping for a given point *P* we proceed as follows. First we will identify the cell *C* to which P belongs. We know that *C.full* list contains all the queries that fully cover *C*, thus *P* is relevant to each and every query in this list. List *C.part* contains queries that partially cover *C* and *P* may or may not be relevant to each of these queries. Each query in this list is checked against *P*. Each query that covers *P* is added to the point-to-query mapping for *P*. For a set (array) of points for which the complete point-to-query mapping is needed, the following step is applied to each point in turn:

```
for (each point P in array)
{
  index->processPoint(P, full_Qlist_for_P, part_Q_list_for_P);
}
```

That is for each point the result is given in two lists. These lists contain all the queries the point is contained in. List *full_Qlist_for_P* is the same list as *C.full*, where *C* is the cell to which *P* belongs. List *part_Q_list_for_P* is constructed from *C.part* list as described above. It should be noted that it is not necessary to duplicate the list *C.full*. The pointer *full_Qlist_for_P* can simply to point to *C.full*. Thus part of the answer is pre-computed and constructing *full_Qlist_for_P* takes O(1) time. If it is desired that a single list be returned, a wrapper 'set' class can easily be constructed. This class would encapsulate the lists and handle each appropriately and support operations such as enumeration, duplication and destruction. In order to compute both mappings (point-to-query & query-to-point), the following algorithm is applied:

```
for (each point P in array)
{
  index->processPoint(P, full_Qlist_for_P, part_Q_list_for_P);

  for (each query Q in full_Qlist_for_P)
    point_list_for_Q.addPoint(P);

  for (each query Q in part_Qlist_for_P)
    point_list_for_Q.addPoint(P);
}
```

At the end of this process, the list *point_list_for_Q* will contain all the points contained in query *Q*.

4 Improving Cache Hit Rate: Z-Sort Optimization

The algorithms presented above for computing the mappings iterate through the array of points. For each point, the *full* and *part* lists of its cell are accessed. The algorithm simply processes points in sequential order in the array. Consider the example shown in Figure 4-1. The order in which the points appear in the array is shown on the left of the figure in the "Unsorted Point Array". In this example the lists pointed to by Cell (0,0) will be accessed for processing point P_2 and then later for processing point P_n.

However if we re-order points in the array such that points that are close together in our 2D domain are also close together in the point array as the array on the right labeled "Z-Sorted Point Array" shown in Figure 4-1. With this ordering, point P_2 will be analyzed first and therefore Cell(0, 0) and its lists will be processed. Then point P_n will be analyzed and Cell(0, 0) and its lists will be processed again! In this situation everything relevant to Cell(0, 0) is likely to remain in the CPU cache after the first processing and will be reused from the cache during the second processing. The speed up effect is achieved also because points that are close together are more likely to be covered by the same queries than points that are far apart, thus queries are more likely to be retrieved from the cache rather than from main memory.

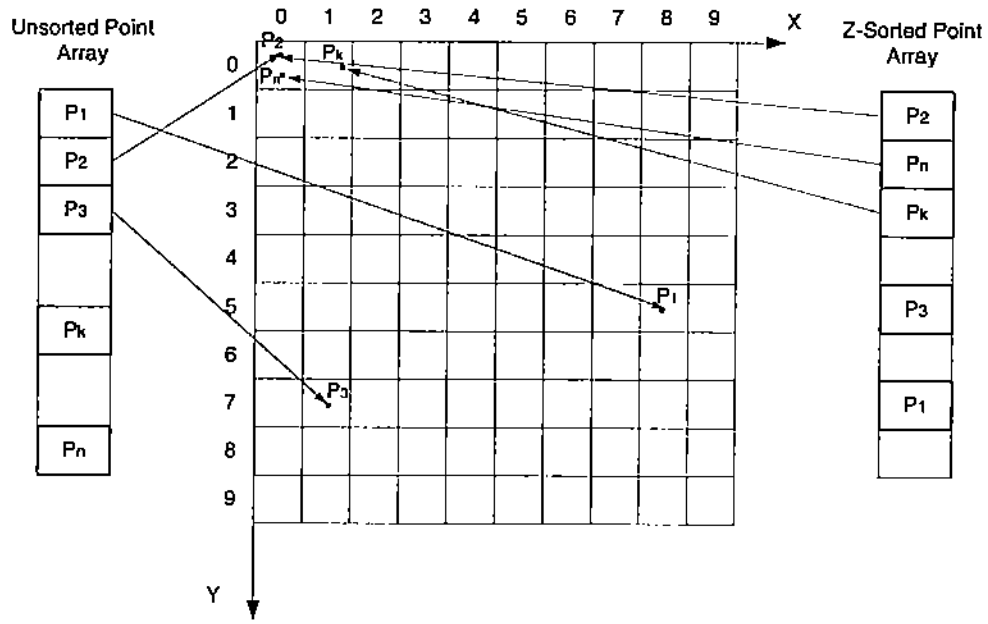

Figure 4-1 Example of sorted and un-sorted Object Arrays

Sorting the points to ensure that points that are close to each other are also close in the array order can easily be achieved 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. The z-order is a recursive graph that can be used to order regions to an arbitrary degree of resolution.

The ordering is defined recursively. First we divide the domain into four equal parts and name these parts "1", "2", "3", and "4" as depicted on Figure 4-2. Next we re-order points in the array such that all points which belong to part 2 are placed after all point from part 1. Also all points from part 3 are placed after all points from part 2 and all points from part 4 are placed after all points from part 3. This process is applied recursively to part 1, part 2, part 3 and part 4. The bottom of the recursion is reached when there is one or no points left in currently processed part. If two points in the array are very close, this could lead to deep recursion. To avoid this, we apply an additional condition: if the size of currently processed part is less some value do not divide any further. Figure 4-3 illustrated the z-ordering for a sample set of points.

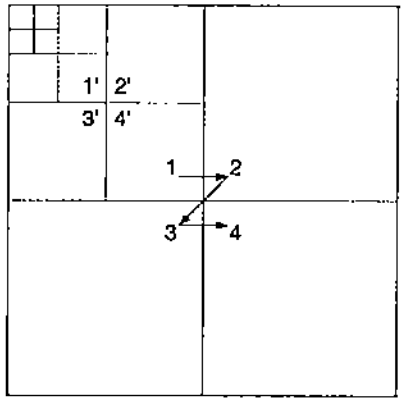

Figure 4-2 The Z-order

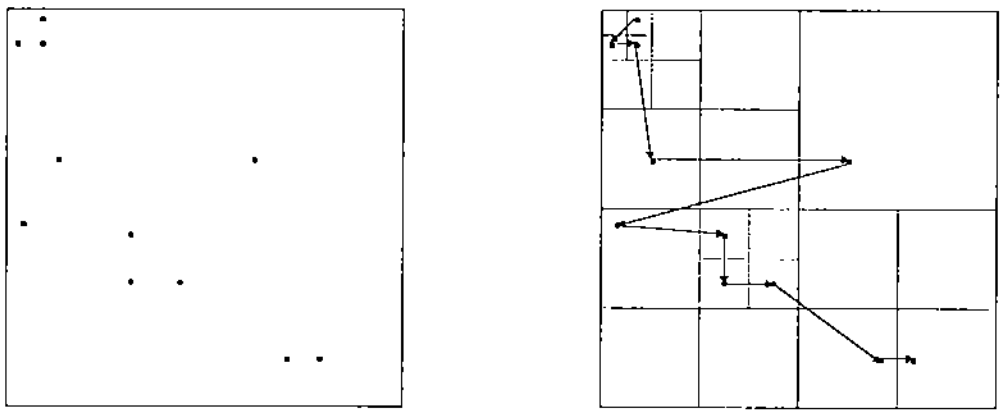

Figure 4-3 (a) Point s to be sorted (b) Z-sorted points.

Reordering the points using a space-filling curve significantly improves the performance of the main memory algorithm, as will be seen in Section 6. Although the advantage of having a sorted array are clear, there is a price to be paid for performing the sorting. If we have to sort the array each time before generating the mapping the cost of sorting could potentially negate the gains. However, there for the case of moving objects, it is not necessary to sort the array each time. This is because objects do not travel very large distances in a very short period of time. Objects are likely to remain close to their current locations at least for a short period of time. For the case of humans, this is even more likely (e.g. staying at home during the night and in an office during

the day). Thus the sorting will remain more or less correct for large durations of time. The cost of sorting can therefore be amortized over several computations of mappings.

5 Navigation queries

The periodic addition or deletion of queries can easily be handled using the CELL-Index as discussed earlier. Using CELL-Index structure we are also able to handle navigation queries very effectively. A navigation query is a moving query that follows a certain path. The simplest implementation of a navigation query is to remove the query from index, adjust its coordinates, and add this query to the index again. The following is a part of our C++ code responsible for the simple movement:

```
index->removeQuery(q);
q->x += stepX;
q->y += stepY;
index->addQuery(q);
```

As will be seen from the experimental results, this method is preferable when the query is moving fast. For queries that are not moving very fast we propose two alternative techniques for handling navigation queries: **OPT-1** and **OPT-2.**

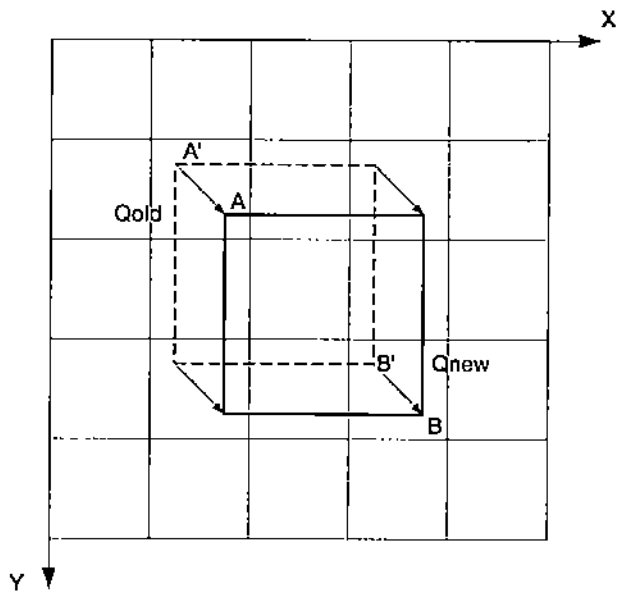

Figure 5-1 Example of Query Navigation

The first optimization, **OPT-1**, is as follows. A cell can be in three states regarding query Q: it can be fully covered by Q, it can be partially covered by Q, or there is no overlap with Q. Please refer to Figure 5-1. It is easy to see that if the coordinates of the two diagonal corner points A and B of the new query location remain within the same cells boundaries as the original query and the states of these two cells

regarding Q do not change, then no modification to the CELL-Index is necessary. In other words, if neither of the two corners crosses a cell boundary (note that being on a cell boundary is equivalent to crossing it) then no update to the CELL-Index lists is necessary. Such a query can be moved simply by adjusting its coordinates.

The second optimization, **OPT-2**, is as follows: If a query move does not satisfy the conditions for OPT-1, it is necessary to update the CELL-Index. The second optimization tries to minimize this update. If the query is large so that it covers many cells and it moves slowly there will be a region of cells R containing query Q such that the state of the cells won't change after the move, see Figure 5-2. No processing is necessary for the cells in that region. OPT-2 identifies this region, removes Q from lists of the cells that contain Q except for the cells in R, updates Q coordinates, adds Q to the proper lists of appropriate cells except for the cells in R.

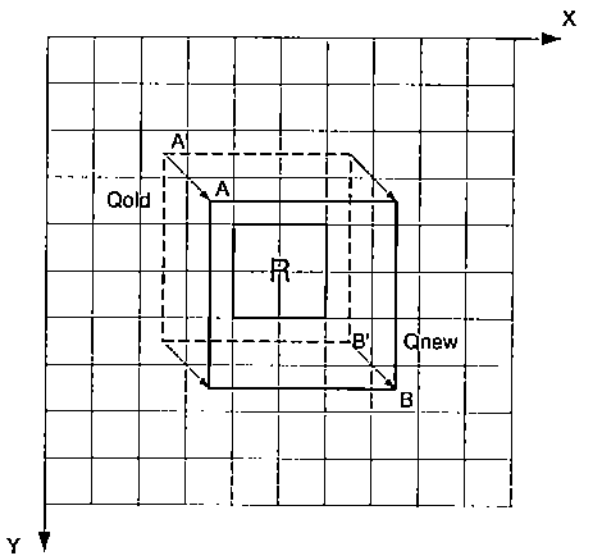

Figure 5-2 Example of OPT-2 Update

The continuous addition and deletion of navigation queries from lists can be an expensive operation if there are long lists (e.g. if there is a region with many queries). This turns out not to be a problem since entries to the lists are added at the beginning. When a query is added to a list it becomes its first element and thus can be removed in O(1) time. If in the system there are many continuous queries and several navigation queries, then navigation queries will be at the beginning of lists of the cells in the CELL-Index. Thus it requires the same amount of time to remove the navigation queries from the index regardless of the number of continuous queries already in the index. Since adding a query to a list is always O(1), we have the following important property: *performance results for navigation queries are*

independent of any characteristics of the continuous queries and points. As will be seen in Section 6, performance results are quite impressive for all three methods. The algorithm employing both optimizations is the most effective technique among the three. Please refer to the next Section for the results.

6 Experiments

In this section we present the performance results for our proposed techniques. 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.

6.1 Experiment setup

The results are based upon an actual implementation of the proposed algorithms. In all our experiments we used Pentium III, 1GHz machine with 2GB of memory running Red Hat Linux 7.1. The machines has 32KB of Level 1 Cache (16K for instructions and 16K for data) and 256KB Level 2 Advanced Transfer Cache. Even though 2MB of main memory was available, the amount of memory required for the data structures was much smaller. In the worst case 391MB was used. Of course, the amount of memory required to hold the resulting mappings varies with the relative placement of objects and queries and is not included in the 391MB – this requirement is the same for any solution to this problem. The code was written in C++ and compiles on UNIX or Windows machines. We used the following commands to compile our code:

```
gcc -O4 -Wall -c *.cpp
```

The clarity of our C++ code was of much greater importance to us than the speed of the program, so that sometimes we introduced unnecessary indirections. We feel that the results can be further improved by careful re-writing of the code in C.

Moving objects were represented as points distributed on [0, 1] x [0, 1] domain. The number of points ranges from 10,000 to 1,000,000. Range-queries were represented as squares with sides ranging from 0.00001 to 0.05, with a size of 0.01 for majority of the experiments. For distributions of points and queries we considered two major cases: a) points and queries distributed uniformly on [0, 1] x [0, 1]; and b) skewed (or clustered) distribution. In the second case we considered datasets with five clusters. The points and queries are evenly distributed among the clusters. Point and queries are distributed normally with a standard deviation of 0.05. In the majority of our experiments CELL-Index was chosen to consist of a grid with 1000x1000 cells.

The testing proceeds is as follows. First queries and points are generated and put into arrays. The algorithms make no assumptions about the speed of objects and have the important property *that the algorithm is applicable to objects moving with arbitrary velocities and on arbitrary paths.*

Next, if needed, the Z-Sort procedure applied to the objects and/or queries. Then the CELL-Index is initialized and the queries are added to it. Following these steps the different algorithms are executed depending on the experiment. The time needed to compute full point-to-query (or both) mapping(s) *(processing time)* is measured.

6.2 Impact of the number of points and queries on processing time.

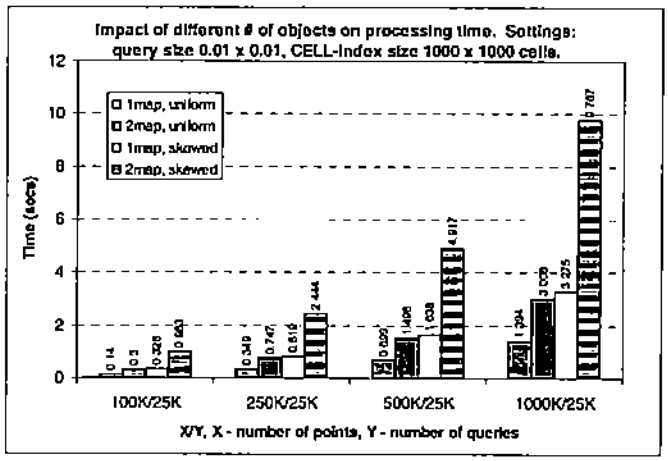

Figure 6-1 Scalability with number of objects.

First we study the scalability of the technique as the number of moving objects increases. In Figure 6-1 the processing time in seconds needed to compute the mapping(s) for different numbers of **objects** (points) is shown. Four experiments with 100K, 250K, 500K, and 1M points respectively are reported. The number of queries was fixed at 25K. For each setting, four bars representing the following four cases are shown:

1. 1-mapping for uniform data.
2. 2-mappings for uniform data.
3. 1-mapping for skewed data.
4. 2-mappings for skewed data.

The first thing that should be noted is that the algorithm is very fast – it can compute 1-mapping for uniform data with 1M objects and 25K queries in just 1.394 secs. The second property that should be noted is that the processing time for all four cases roughly doubles as the number of points doubles. The processing time scales nearly linearly as a function of the number of points.

The time required for computing 2 mappings is, as expected, more than the time required for computing a single mapping. The time for computing mappings for skewed data is greater than that for uniform data. This is largely due to the fact that this is a very pessimistically chosen set of queries and data. Due to the clustering of queries, the average length of the queue lists (not including the empty lists) is expected to be large. This is because cells that cover the clusters will have lots of queries covering them. At the same time, most of the objects (since they are also distributed using the same clusters) will also map to these cells with large queues. Hence the time required for processing most points will require traversals of long lists of partially covering queries.

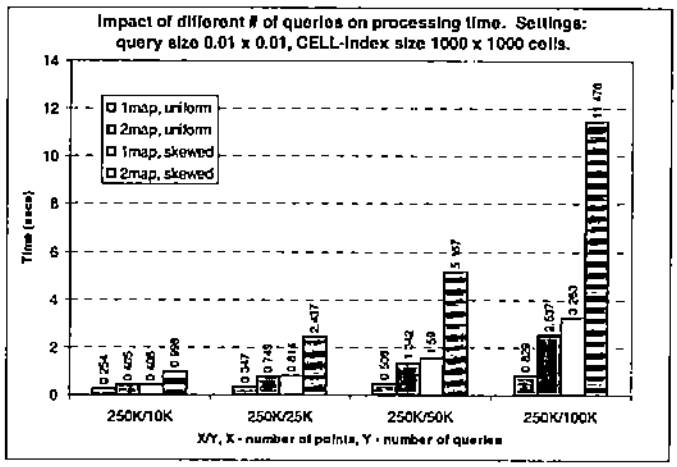

Figure 6-2 Scalability with number of Queries

Next we study the scalability with number of queries. In Figure 6-2 the results for various numbers of queries are presented. The graph shows the processing time in seconds needed to compute the mapping(s) where the number of **queries** change, taking values 10K, 25K, 50K, and 100K while the number of points is fixed at 250K. It can be seen that the algorithm scales linearly with the number of queries – the processing time for all four cases roughly doubles as the number of **queries** doubles. As with the earlier experiment, the algorithm is very effective in computing the mappings for both uniform and skewed data.

We now study the behavior of the algorithms as both objects and queries are varied. In Figure 6-3 the results with the number of queries fixed to be 10% of the number of points is shown. As seen in the

Figure, the processing time needed for all the four cases increases roughly four times as the numbers of points and queries are doubled simultaneously. The increase is once again roughly linear. The fourth bar representing 2-mappings for skewed data for 1M points and 100K queries is absent because the two mappings required too much memory space to store. Note that if indeed an application needs this data, it can be streamed to disk or should, if possible, be consumed concurrently to avoid running out of memory.

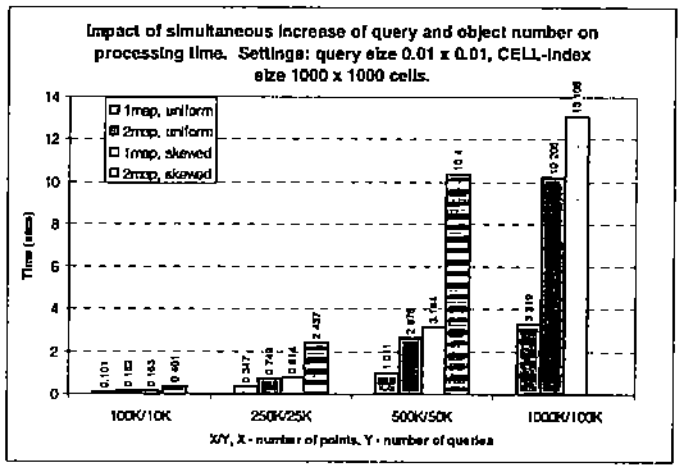

Figure 6-3 Scalability with numbers of objects and queries.

6.3 Impact of CELL-Index size.

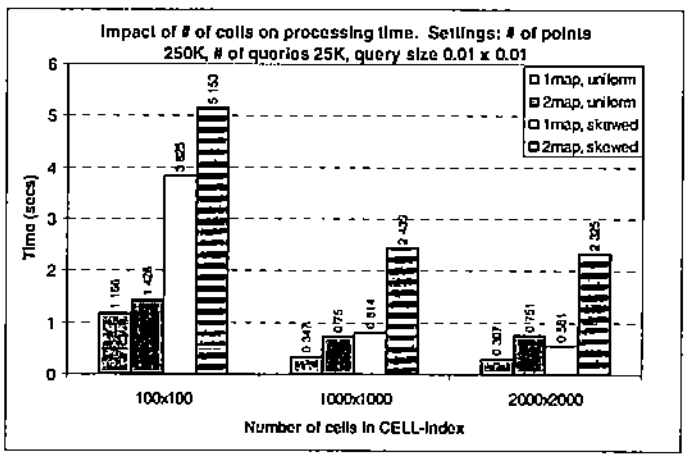

Figure 6-4 Impact of number of cells on processing time.

In this experiment we study the impact of the number of cells in the CELL-Index. Figure 6-4 presents the processing time needed with CELL-Index sizes 100x100, 1000x1000, and 2000x2000 cells. There is a substantial increase in performance as we move from 100x100 cells to 1000x1000 cells. The increase is minor when we move from 1000x1000 to 2000x2000 cells for our case of 250K points and 25K. For a bigger data set (e.g. 1M / 100K) this increase will also be noticeable. Increasing the number of cells has the effect of reducing the average number of queries for a cell thereby reducing the time required for generating the mapping for each given object. Increasing the number of cells is beneficial for computing the mapping, however, it impacts changes to queries negatively. The next experiment presents the trade-off between processing time and time needed to add/remove queries from index depending on the CELL-Index size.

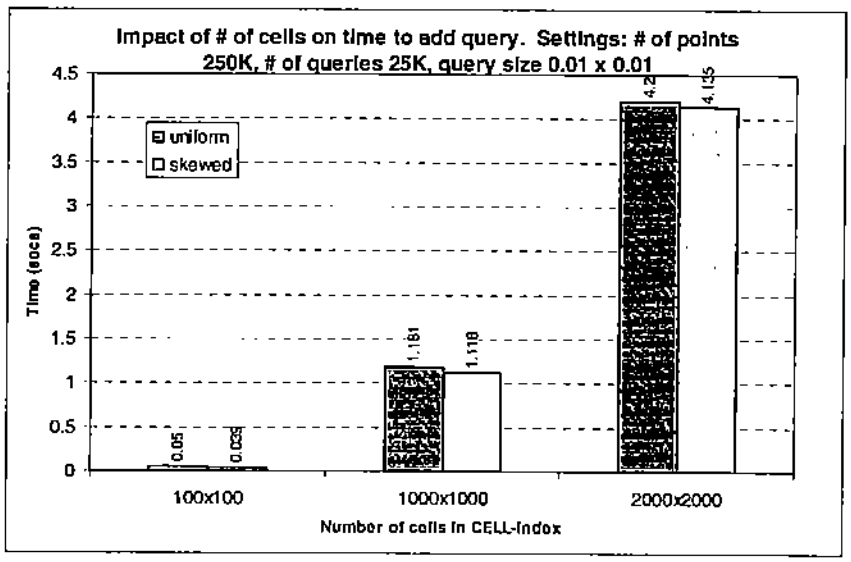

Figure 6-5 Time to add queries for various numbers of cells.

Figure 6-5 presents the time required to add all 25K queries to the index for different numbers of cells: 100x100, 1000x1000, and 2000x2000 cells. It can be seen that the time needed to add the same 25K queries increases four times when we go from 1000x1000 cells to 2000x2000 cells. This can be explained by the fact that the same query now covers four times more cells. Thus time to add queries scales linearly as a function of the number of cells. Notice that these times are for adding 25,000 queries – which is not expected to happen often for most environments. Therefore, a finer grid is preferable to improve the time for computing the mapping. The size of the grid that should be used is therefore limited by the amount of memory available. A finer grid implies more pointers. Thus the choice of grid size should be made based, number of queries, and available memory size. The number of objects is not an issue in this choice. The

available memory size if affected by the amount of memory that is needed to hold the resulting mappings. If the results are streamed to disk or consumed concurrently, then the amount available can be larger – this depends upon the application for which the mappings are produced.

An interesting property to note is that in this case time to add skewed query is less then time to add uniform query, while they should be the same. This has to do with the placement of queries in the query array – the way queries are generated. We place one cluster after another in query array for skewed data. That is why in case of skewed distribution queries that are close to each other in domain tend to be closer to each other in query array than in uniform case. Thus queries that are close to each other are processed closed to each other in time. More on this in Z-Sort result section.

6.4 Impact of query size on processing time

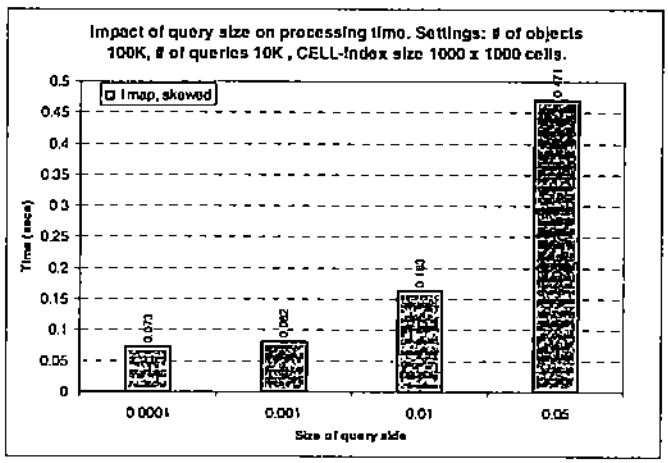

Figure 6-6 Impact of query size on processing time.

Figure 6-6 illustrates the processing time needed when query side size take values of 0.0001, 0.001, 0.01, and 0.05 for 100K points and 10K queries. These values demonstrate cases when a query is smaller then a cell (0.0001), equal in size to a cell (0.001) and bigger then a cell (0.01 and 0.05). The difference between 0.0001 and 0.001 cases is minor because the processing procedure needs to execute approximately the same amount of code needed to determine which side must be added to which list etc. For both cases none of the queries will fully cover any cell. As the size of the queries gets larger the time required to compute the mapping increases. This is expected because for larger query coverage, more cells will be covered by each query, resulting in longer query lists.

6.5 Time to add/remove queries to/from CELL-Index

We now study the efficiency of modifying queries. The results shown in Figure 6-7 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 is done very efficiently with the CELL-Index. 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 ½ for each query. Therefore we see that even significant changes to the query set can be effectively handled by the CELL-Index approach.

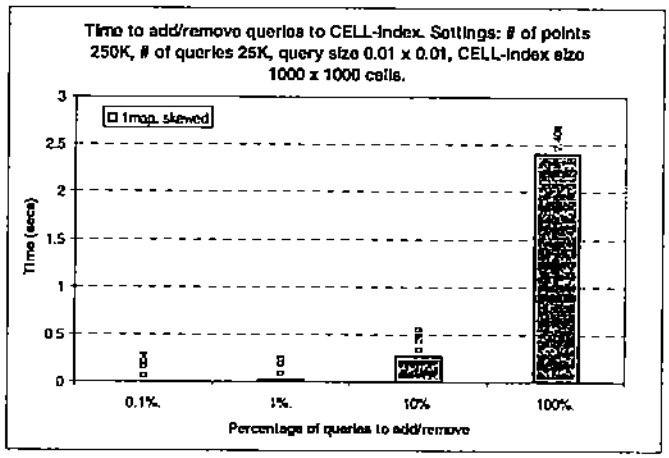

Figure 6-7 Adding and deleting queries.

6.6 Z-Sort

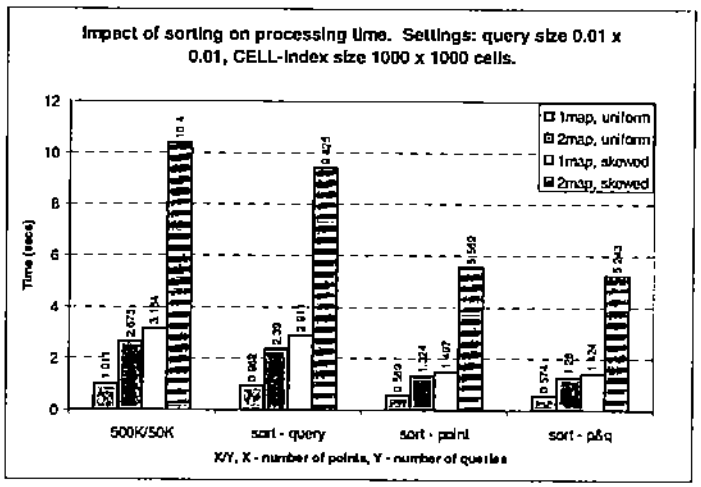

Figure 6-8 Effectiveness of Z-Sorting.

Figure 6-8 illustrates effect of Z-Sort technique on processing time. 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 close to each other will tend to reuse information stored in the cache rather then retrieving it from main-memory. This graph shows the processing time needed for 500K points and 50K queries when no data is sorted, only queries are sorted, only points are sorted, and both query and points are sorted. From the results, we see that sorting points decreased the processing time by roughly 50%, whereas sorting the queries had only a limited effect. Sorting the queries is beneficial for the loading of queries into the index, but does not significantly affect the computing of the mapping. For larger data set (e.g. 1M / 100K) the coefficient of improvement when sorting points was found to be around 3.

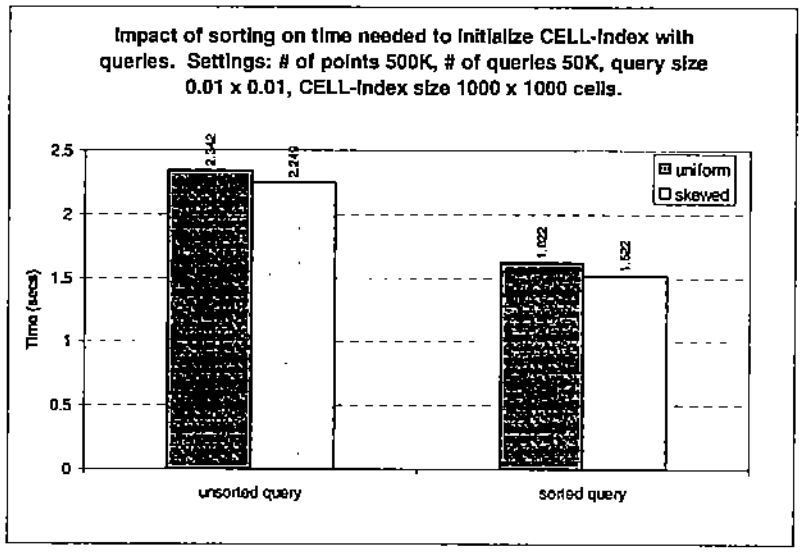

Figure 6-9 Impact of sorting queries on CELL-Index creation.

Figure 6-9 demonstrates the effect of sorting on time needed to initialize the index with all its queries. That is what time it takes to add all queries to the index. As can be seen sorting of queries has significant impact. It also can be seen that the time for skewed distributions is less than the time for uniform even for sorted array of queries. This can be explained by the fact that in the skewed case queries are closer to each other in the array than in the uniform case and are thereby more likely to benefit from cache hits. Computing the sorting is also very fast: in the worst case of 1000K skewed objects, the sorting took 4.4 secs; for 100K skewed queries, the sorting took 0.55 seconds.

6.7 Navigation

In the experiments devoted to navigation queries we evaluate the time needed for a navigational query to make 90 steps on the line from a point close to the origin having the step in vertical direction stepY always be equal to stepX/2, while stepX is fixed and for different cases takes values of 0.00001, 0.0001, 0.001, and 0.01. The direction of the movement is demonstrated in Figure 6-10. For fairness of comparision the same movement of the query is chosen for all the experiments. The actual path of the query is less important for performance than the rate at which the query crosses cell boundaries. Hence the size of the query and the speed of travel are more important parameters than the actual location and path of the query.

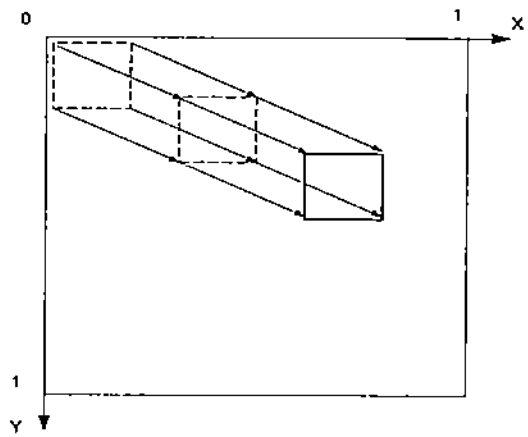

Figure 6-10 Movement of Navigation query

We measured the time needed to complete 100 runs. Each run includes adding a query to the index, performing 90 steps using simple or smart algorithm, and removing the query from the index. Each of the graphs in Figure 6-11, Figure 6-12, and Figure 6-13 demonstrates the time required by the two approaches for different stepX values while query size is fixed per graph. For Figure 6-11 the query size is fixed at 0.01 x 0.01, which demonstrates a case when each query is bigger than a cell size. In Figure 6-12 the query size is 0.001 x 0.001 and is equal to the cell size. In Figure 6-13 each query size is fixed at 0.0001 x 0.0001 and is smaller than a cell size. The *simple* algorithm simply removes the query from its old location and adds it at its new location. The *smart-1* algorithm applies optimization **OPT-1**, and the *smart-2* algorithm applies both **OPT-1** and **OPT-2** optimizations described in Section 5.

As can be seen from the three figures, the optimizations give better performance in all cases except when the query speed is so high that the query moves a distance greater then its own size in each step. We do

not mention how many points and how many queries were used in the experiments. The number of points is obviously irrelevant here. But it is interesting that the number of queries is irrelevant too. This happens because adding a query will place it at the beginnings of the cell lists. Thus removing it is an O(1) operation since the first element checked in the list will be our query and the algorithm need not proceed further. *An important property of our algorithm is that the performance of navigation queries does not depend on any characteristics of the points or the queries.*

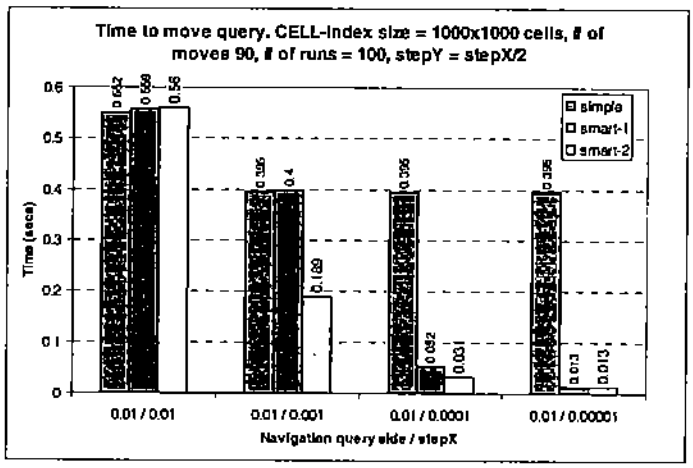

Figure 6-11 Navigation Queries with Side 0.01

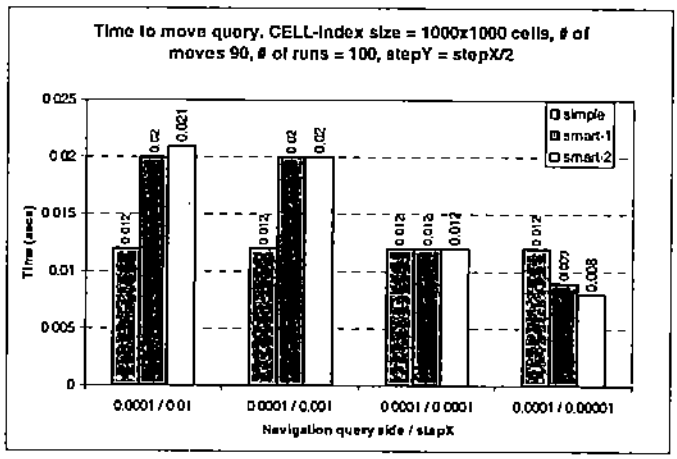

Figure 6-12 Navigation Queries with Side 0.001

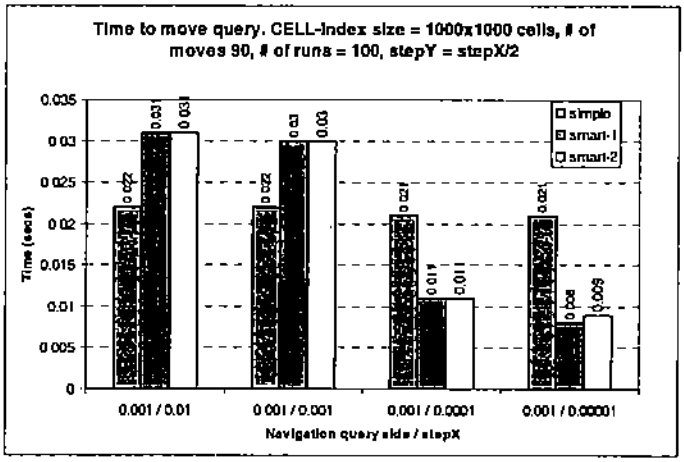

Figure 6-13 Navigation Queries with Side 0.0001

7 Summary

We addressed the general problem of evaluating continuous range queries over moving objects in main memory. The techniques developed can be used as a basis for solving a number of problems that arise in location-aware computing as well as for some streaming data environments. Due to the need for near real-time responses and the continuous changes in the location of the objects this problem is ideally suited for main memory evaluation (keeping a disk-based index updated as objects move is very expensive). To the best of our knowledge, this is the first study to investigate a main memory solution for evaluating continuous queries.

We presented a new main memory index structure called the CELL-Index that is highly effective in computing the mappings of objects to queries and queries to objects. To further improve performance we introduced the Z-sort optimization that results in better cache hit rates – an important performance factor for main memory computing. Although we expect that the set of continuous queries does not change rapidly, our experimental results show that the approach is able to handle large changes in the query set very efficiently. Moreover, for navigation queries that continuously change their query regions, we developed optimizations that enable quick update times in the index.

Based on our extensive experimental results, we demonstrated the very fast execution times of our proposed approach even for large numbers of queries and moving objects. The algorithms scale almost linearly with the numbers of moving objects and queries. An important characteristic of the technique is that it imposes no constraints on the movement of objects. Similarly, no restrictions are made on the movement of navigation queries. The techniques presented solve an underlying problem of computing mappings with no assumptions about the movement of objects or queries, enabling them to be useful for

several applications. It is interesting to note that the number or distribution of continuous queries or moving objects does not affect the performance of navigation queries. The Z-Sort optimization was seen to improve the performance of computing the mappings by as much as factors of 2 or 3, and to reduce the time required to add queries by as much as a factor of 2.

Overall we conclude that the proposed CELL-Index, Z-Sort, and other optimizations are highly effective means for computing continuous range queries over moving objects. We are currently evaluating the performance of our techniques for other applications such as continuous spatial joins (e.g. the ε-join). Details of the results will be reported in the final version.

8 References

[1] S. Acharya, M. J. Franklin, and S. Zdonik. Disseminating updates on broadcast disks. In Proceedings of the 22nd International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India, pages 354-365, Los Altos, CA 94022, USA, 1996.

[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. 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.

[5] 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.

[6] Philip A. Bernstein, Michael L. Brodie, Stefano Ceri, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph M. Hellerstein, H. V. Jagadish, Michael Lesk, David Maier, Jeffrey F. Naughton, Hamid Pirahesh, Michael Stonebraker, Jeffrey D. Ullman. The Asilomar Report on Database Research. SIGMOD Record, 27(4), pages 74-80, 1998.

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

[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] 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.

[13] Tomasz Imielinski, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. In Proc. of the International Conference on Management of Data, pages 25-36. ACM Press, May 1994.

[14] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pages 139-

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

[16] H. Koshima and J. Hoshen. Personal locator services emerge. IEEE Spectrum, 37(2):41-48, 2000.

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

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

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

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

[21] P. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. In Proc. of the 26th Int. Conf. on Very Large Databases (VLDB), Cairo, Egypt, 2000.

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

[23] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pages 475-486, Dallas, TX, 2000.

[24] 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.

[25] 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.

[26] H. Samet. The Design and Analysis of Spatial Data Structures. Addison-Wesley, 1990.

[27] A. Y. Seydim, M. H. Dunham, and V. Kumar. Location dependent query processing. In Proc. of the 2nd ACM international workshop on Data engineering for wireless and mobile access, pp 47-53, 2001

[28] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In Proc. of the 14th Int. Conference on Data Engineering (ICDE'97), pages 422-432, 1997.

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

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

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

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

[33] 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.

[34] 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.

[35] 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.

[36] J. M. Zagami, S. A. Parl, J. J. Bussgang, and K. D. Melillo. Providing universal location services using a wireless E-911 location network. IEEE Communications Magazine, April 1998.

[37] 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, 1994.