EFFICIENT QUERYING OF CONSTANTLY EVOLVING DATA

A Thesis

Submitted to the Faculty

of

Purdue University

by

Dmitri V. Kalashnikov

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2003

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Kalashnikov, Dmitri V. Ph.D., Purdue University, May, 2003. Efficient Querying of Constantly Evolving Data. Major Professor: Sunil Prabhakar.

This thesis addresses important challenges in the emerging areas of sensor (streaming data) databases and moving objects databases. It focuses on the important class of applications that are characterized by (a) constant change in the data values; (b) long-running (continuous) queries that have to be repeatedly evaluated as the data changes; (c) inherent imprecision in the data; and (d) need for near real-time results. The thesis addresses the scalability and performance challenges faced by these applications. The first part of the thesis studies the problem of scalable efficient processing of continuous range queries on moving objects. We introduce two novel highly scalable solutions to the problem: a disk-based technique called Velocity Constrained Indexing (VCI) and an in-memory technique called grid indexing. VCI is a technique for maintaining an index on moving objects that allows the index to be useful without constantly updating it as the data values change. For in-memory settings, we show the superiority of our grid indexing solution to other methods. The second part of the thesis covers the problem of similarity joins for low- and high-dimensional data. Two new similarity join algorithms are introduced: the Grid-join is for low-dimensional data and the EGO*-join is for high-dimensional data. Both algorithms show substantial improvement over the state of the art similarity join algorithms for low- and high-dimensional domains. Finally, the third part of the thesis presents an analysis and novel solutions of the important problem of handling the uncertainty inherent in the environments with constantly changing data. Probabilistic queries are introduced and a classification of queries is developed based on the nature of query result set.

Algorithms are provided for solving typical probabilistic queries from each class. We show that, unlike standard queries, probabilistic queries have a notion of quality of answer. We introduce several metrics for measuring the quality as well as various update policies for improving it.

1. INTRODUCTION

The thesis research addresses important challenges in the emerging areas of sensor (streaming data) and moving objects databases. It focuses on the important class of applications that are characterized by (a) constant change to the data values; (b) long-running continuous queries that have to be repeatedly evaluated as the data changes; (c) need for near real-time results; and (d) inherent imprecision in the data. The thesis addresses the scalability and performance challenge inherent to such applications: all solutions have to scale to large problem sizes, such as millions of sources of constantly changing data and tens of thousands of continuous queries. The contribution of this thesis is summarized below.

1.1 Scalable algorithms for continuous range queries on moving objects

The combination of personal locator technologies, global positioning systems, and wireless and cellular telephone technologies enables new location-aware services, including location and mobile commerce (L- and M- commerce). Current location-aware services allow proximity-based queries including map viewing and navigation, driving directions, searches for hotels and restaurants, and weather and traffic information. To support many of their services, the majority of such applications must be able to handle a fundamental type of continuous query – the range query. Moving object environments are characterized by large numbers of moving objects and numerous concurrent continuous queries over these objects. Efficient evaluation of these queries in response to the movement of the objects is critical for supporting acceptable response times.

Chapter 2 presents a novel technique for the efficient and scalable evaluation of multiple continuous range queries on moving objects called Velocity Constrained Indexing (VCI) and compares it to the Query indexing technique. VCI takes advantage of the maximum possible speed of objects in order to delay the expensive operation of updating an index to reflect the movement of objects. VCI is an R-tree like index, maintained on moving objects, with maximum velocity information in every node. In contrast to an earlier technique that requires exact knowledge about the movement of the objects, VCI does not rely on such information. We show that Query Indexing does not handle the arrival of new queries efficiently while Velocity constrained indexing, on the other hand, is unaffected by changes in queries. A combination of Query Indexing and Velocity Constrained Indexing enables the scalable execution of insertion and deletion of queries in addition to processing ongoing queries. Several optimizations are presented in Section 2.4.1 and Section 2.5.1. A detailed experimental evaluation is presented in Section 2.6. The experimental results show that the proposed schemes outperform the traditional approaches by almost two orders of magnitude.

1.2 In-memory evaluation of continuous range queries on moving objects

It is becoming increasingly clear that the traditional database paradigm of data being stored completely on disk is not acceptable for many applications in the face of current and future rates of data generation. A prime example is that of sensor environments such as network routers that are constantly producing data that cannot all possibly be stored in a database. Moreover the data needs to be processed as it is being generated. In addition, market forces are resulting in computer architectures with abundant amounts of main memory. Sensor database research demonstrates the shift in mentality where currently all processing is assumed to be carried out in main memory and disk-based algorithms are avoided.

Chapter 3 presents evaluation of 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. One of the most important aspects of this work is that no constraints are imposed on the speed or path of moving objects and on the fraction of objects that can move at any time instant (common restrictions of other related work) making the proposed approach applicable for more general settings of sensor databases for the case where two dynamic attributes need to be queried. For the disk-based case, it is well-known that R-trees provide good index performance. Researchers have investigated main-memory versions of R-trees. However it was discovered that the use of entirely different index structures can be more appropriate in the main-memory setting. In particular, Section 3.3 presents a detailed analysis of a cache-conscious index which shows the best results for both skewed and uniform data. Experimental evaluation established that indexing queries using the cache-conscious index yields orders of magnitude better performance than other index structures such as memory optimized disk-based indexes. A sorting-based and clustering optimizations were also developed which significantly improved the cache hit ratio.

1.3 Similarity joins for low- and high-dimensional data

A similarity join operation for two sets containing multidimensional points and fixed epsilon parameter finds all pairs of points from different sets which are within epsilon distance from each other. This operation is used in multimedia databases, data mining, location-based applications, and time-series analysis. The efficient processing of similarity joins is important for a large class of applications. The dimensionality of the data for these applications ranges from low to high. Chapter 4 introduces two new spatial join algorithms; one aimed for low-dimensional data (2–6 dimensions), the other for high-dimensional data. Section 4.4 studies their performance in comparison to the state of the art algorithm EGO-join, and the RSJ algorithm. Through evaluation the domain of applicability of each algorithm is explored and recommendations are provided for the choice of join algorithm depending upon the dimensionality of the data as well as the critical epsilon parameter. Section 4.4.1 points out the

significance of the choice of this parameter for ensuring that the selectivity achieved is reasonable. The proposed join algorithms significantly outperform the state of the art join algorithm (EGO-join) across various domains of applicability.

1.4 Handling data uncertainty

Many applications employ sensors for monitoring entities such as temperature and wind speed. A centralized database tracks these entities to enable query processing. Due to continuous changes in these values and limited resources (e.g., network bandwidth and battery power), it is often infeasible to store the exact values at all times. A similar situation exists for moving object environments that track the constantly changing locations of objects. In this environment, it is possible for database queries to produce incorrect or invalid results based upon old data. The impact of imprecision is data is an important problem for emerging applications. Chapter 5 addressed the problem of evaluation of certain types of queries in presence of imprecision in data. It defines various quality metrics for query answers. Query dependent update policies were developed in order to improve the quality of answers.

1.4.1 Probabilistic queries

Although it is infeasible to store the exact values of all data items at all times, if the degree of error (or uncertainty) between the actual value and the database value is controlled, one can place more confidence in the answers to queries. More generally, query answers can be augmented with probabilistic estimates of the validity of the answers. The imprecision in answers to the queries is an inherent property of these applications due to uncertainty in the data, unlike the techniques for approximate query processing that trade accuracy for performance.

Chapter 5 presents the the evaluation of probabilistic guarantees for a broad class of database queries including range and nearest-neighbor queries for moving objects, and several aggregate queries for general sensor databases. A classification of queries was made based upon the nature of the result set. For each class, algorithms for

computing probabilistic answers were developed. The algorithms were defined in terms of a generic model of uncertainty.

1.4.2 Quantifying query result quality

This research provides an insight into the important issue of measuring the quality of the answers to probabilistic queries. Chapter 5 presents an initial study of the issue of quantifying the quality through metrics such as entropy. A related problem is that of improving the quality of an answer by selectively reducing the uncertainty of the underlying data under constrained resources. The initial work has resulted in the development of heuristics for improving query quality.

The rest of the thesis is organized as follows. Chapter 2 introduces "query indexing" and discusses Velocity Constrained Indexing. In Chapter 3 various in-memory query indexing algorithms are evaluated. Similarity joins are covered in Chapter 4. Chapter 5 explores methods of dealing with uncertainty in data. Finally, Chapter 6 concludes the thesis.

2. CONTINUOUS RANGE QUERIES ON MOVING OBJECTS

2.1 Introduction

The combination of personal locator technologies [KH00, WL98], global positioning systems [McN, Tru], and wireless [Cor] and cellular telephone technologies enables new location-aware services, including location and mobile commerce (L- and M-commerce). Current location-aware services allow proximity-based queries including map viewing and navigation, driving directions, searches for hotels and restaurants, and weather and traffic information. They include GPS based systems like Vindigo and SnapTrack and cell-phone based systems like TruePosition and Cell-Loc.

These technologies are the foundation for pervasive location-aware environments and services. Such services have the potential to improve the quality of life by adding location-awareness to virtually all objects of interest such as humans, cars, laptops, eyeglasses, canes, desktops, pets, wild animals, bicycles, and buildings. Applications can range from proximity-based queries on non-mobile objects, locating lost or stolen objects, tracing small children, helping the visually challenged to navigate, locate, and identify objects around them, and to automatically annotating objects online in a video or a camera shot. Examples of such services are emerging for locating persons [KH00] and managing emergency vehicles [Ltd99]. These services correspond to queries that are executed over an extended period of time (i.e., from the time they are initiated to the time at which the services are terminated). During this time period the queries are repeated evaluated in order to provide the correct answers as the locations of objects change. We term these queries *Continuous Queries*. A fundamental type of continuous query required to support many of the services

mentioned above is the range query. The range query R, given d-dimensional hyper-rectangle $B^d = [l_1, u_1] \times \ldots \times [l_d, u_d]$, finds all spatial objects x that intersect with B^d: $R(B^d) = \{x : x \cap B^d \neq \emptyset\}$. [1]

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

This chapter develops a novel technique for the efficient and scalable evaluation of multiple continuous range queries on moving objects called *Velocity Constrained Indexing (VCI)*. VCI is compared to another scheme developed by Prabhakar et.al. [PXK+02], called *Query Indexing*. Velocity constrained indexing (VCI) enables efficient handling of changes to queries. VCI allows an index to be useful even when it does not accurately reflect the locations of objects that are indexed. It relies upon the notion of maximum speeds of objects. Our model of object movement makes no assumptions for query-indexing. For the case of VCI, we assume only that each object has a maximum velocity that it will not exceed. If necessary, this value can be changed over time. We do not assume that objects need to report and maintain a fixed speed and direction for any period of time as in [SJLL00]. The velocity constrained index remains effective for large periods of time without the need for any updates, independent of the actual movement of objects. Naturally, its effectiveness drops over time and infrequent updates are necessary to counter this degradation. A combined approach of these two techniques enables the scalable execution of insertion and deletion of queries in addition to processing ongoing queries. We also develop several optimizations for efficient post-processing with VCI through *Clustering*; and for efficient updates to VCI – *Refresh* and *Rebuild*. A detailed experimental evaluation

[1] Sometimes, range query is also defined as $R(B^d) = \{x : x \subset B^d\}$, we use the first definition in our experiments, using the second definition will lead to the same results (since in our tests objects are tiny rectangles).

of our techniques is conducted. The experimental results demonstrate the superior performance of our indexing methods as well as their robustness to variations in the model parameters.

Our work distinguishes itself from related work in that it addresses the issues of scalable execution of concurrent continuous queries (as the numbers of mobile objects and queries grow).

The material on Query-indexing presented in this chapter is a joint work with Walid Aref, Susanne Hambrusch, Sunil Prabhakar, and Yuni Xia.

The rest of this chapter proceeds as follows. Related work is discussed in Section 2.2. Section 2.3 describes the traditional solution and our assumptions about the environment. Section 2.4 presents the approach of Query Indexing and related optimizations. The alternative scheme of Velocity Constrained Indexing is discussed in Section 2.5. Experimental evaluation of the proposed schemes is presented in Section 2.6, and Section 2.7 concludes the chapter.

2.2 Related work

The growing importance of moving object environments is reflected in the recent body of work addressing issues such as indexing, uncertainty management, broadcasting, and models for spatio-temporal data. To the best of our knowledge no existing work addresses the timely execution of multiple concurrent queries on a collection of moving objects as proposed in the following sections. We do not make any assumption about the future positions of objects. It is also not necessary for objects to move according to well behaved patterns as in [SJLL00]. In particular, the only constraint imposed on objects in our model is that for Velocity Constrained Indexing (discussed in Section 2.5) each object has a maximum speed at which it can travel (in any direction).

Indexing techniques for moving objects are being proposed in the literature, e.g., [BGO+96, KTF98] index the histories, or trajectories, of the positions of moving objects, while [SJLL00] indexes the current and anticipated future positions of the moving objects. In [KGT99], trajectories are mapped to points in a higher-dimensional space that are then indexed. In [SJLL00], 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 [PJ99, WSCY99], 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. [TUW98] use quadtrees [Sam90] to index the trajectories of one-dimensional moving points. Kollios et. al. [KGT99] map moving objects and their velocities into points and store the points in a kD-tree. Pfoser et. al. [PTJ99, PJT00] 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 [AAE00] through the use of indexing schemes using external range trees. [WCD+98, WXCJ98] consider the management of collections of moving points in the plane by describing the current and expected positions of each point in the future. They address how often to update the locations of the points to balance the costs of updates against imprecision in the point positions. Spatio-temporal database models to support moving objects, spatio-temporal types and supporting operations have been developed in [FGNS00, GBE+00].

Scalable communication in the mobile environment is an important issue. This includes location updates from objects to the server and relevant data from the server to the objects. Communication is not the focus of this thesis. We propose the use of *Safe Regions* to minimize communication for location updates from objects. We assume that the process of dissemination of safe regions is carried out by a separate

process. In particular, this can be achieved by a periodic broadcast of safe regions. Efficient broadcast techniques are proposed in [AFZ96, AAFZ95, HLAP01, HLL00, HLL01, IVB94, ZFAA94]. In particular, the issue of efficient (in terms of battery-time and latency) broadcast of indexed multi-dimensional data (such as safe regions) is addressed in [HLAP01]. Issues relating to location dependent database querying are addressed in [SDK01]. A excellent review of multidimensional index structures including grid-like and Quad-tree based structures can be found in [TUW98].

In hypertext databases the scope of search might change dynamically, while traditional indexes cover a statically defined region. Indexing in hypertext databases has been studied in [CGM90].

Main memory optimizations of disk-based index structures have been explored recently for B+-trees [RR00] and multidimensional indexes [KCK01]. 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 in-memory algorithm is to efficiently and continuously re-generate the mapping between moving objects and queries. Our in-memory 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 [SJLL00]. 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.

2.3 Moving object environment

2.3.1 Pervasive location-aware computing environments

Figure 2.1 sketches a possible hierarchical architecture of a location-aware computing environment. Location detection devices (e.g., GPS devices) provide the objects

with their geographical locations. Objects connect directly to regional servers. Regional servers can communicate with each other, as well as with the repository servers. Data regarding past locations of objects can be archived at the repository servers. We assume that (i) the regional servers and objects have low bandwidth and a high cost per connection, and (ii) repository servers are interconnected by high bandwidth links. This architecture is similar to that of current cellular phone architectures [SWCD97, Wol00]. For information sent to the objects, we consider point-to-point communication as well as broadcasting. Broadcasting allows a server to send data to a large number of "listening" objects [AFZ96, AAFZ95, HLAP01, HLL00, HLL01, ZFAA94]. Key factors in the design of the system are scalability with respect to large numbers of objects and the efficient execution of queries.

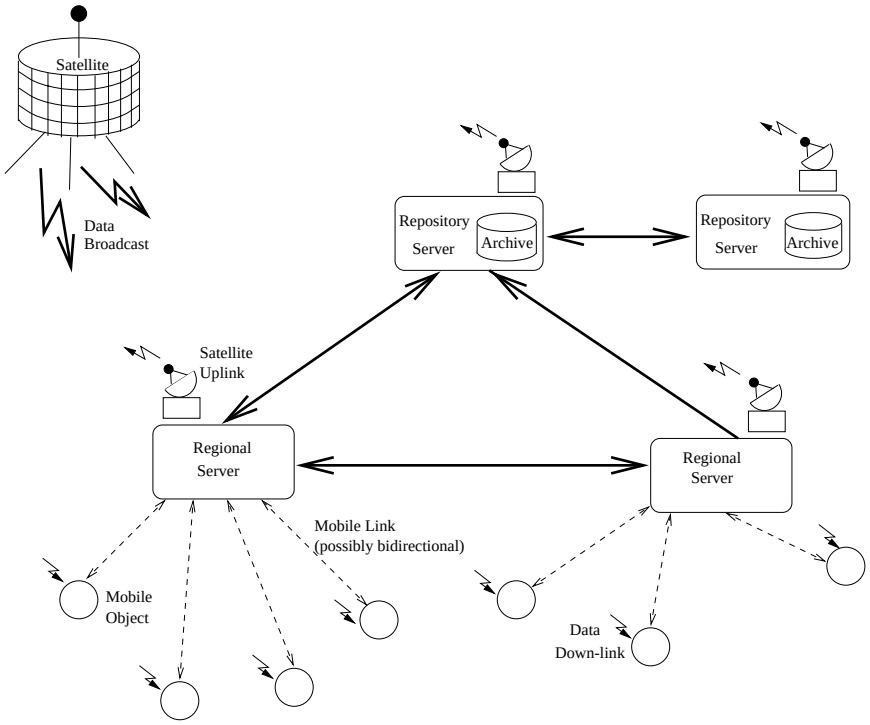


Figure 2.1 Illustrating a location-aware environment

In traditional applications, GPS devices tend to be passive i.e., they do not exchange any information with other devices or systems. More recently, GPS devices

are becoming active entities that transmit and receive information that is used to affect processing. Examples of these new applications include vehicle tracking [Ltd99], identification of closest emergency vehicles in Chicago [Ltd99], and Personal Locator Services [KH00]. Each of these examples represents commercial developments that handle small scale applications. Another example of the importance of location information is the emerging Enhanced 911 (E911) [ZPBM98] standard. The standard seeks to provide wireless users the same level of emergency 911 support as wireline callers. It relies on wireless service providers calculating the approximate location of the cellular phone user. The availability of location-awareness would further enhance the ability of emergency services to respond to a call e.g., using medical history of the caller. Applications such as these, improvements in GPS technology, and reducing cost, augur the advent of pervasive location-aware environments. The PLACE (Pervasive Location-Aware Computing Environments) project at Purdue University is addressing the underlying issues of query processing and data management for the moving object environments [AHKP]. Connectivity is achieved through wireless links as well as mobile telephone services.

2.3.2 Continuous query processing

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

for a timely response for continuous queries, efficient and scalable query execution is paramount.

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

2.3.3 Model

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

Ideally, each query should be re-evaluated as soon as an object moves. However, this is impractical and may not even be necessary from the user's point of view. We therefore assume that the continuous evaluation of queries takes place in a periodic fashion whereby we determine the set of objects that are relevant to each continuous query at fixed time intervals. This interval, or time step, is expected to be quite small

(e.g., in [KGT99] it is taken to be 1 minute) – our experiments are conducted with a time interval of 50 seconds.

2.3.4 Limitations of traditional indexing

In this section we discuss the traditional approaches to answering range queries for moving objects and their limitations. Our approaches are presented in Sections 2.4 and 2.5.

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

Since we are testing for spatial relationships, a natural alternative is to build a spatial index on the objects. To determine which objects intersect each query, we execute the queries on this index. All objects that intersect with a query are relevant to the query. The use of the spatial index should avoid many unnecessary comparisons of queries against objects and thereby we expect this approach to outperform the brute force approach. This is in agreement with conventional wisdom on indexing. In order to evaluate the answers correctly, it is necessary to keep the index updated with the latest positions of objects as they move. This represents a significant problem. Notice that for the purpose of evaluating continuous queries, we are not interested in preserving the historical data but rather only in maintaining the current snapshot. The historical record of movement is maintained elsewhere such as at a repository server (see Figure 2.1).

In Section 2.6 we evaluate these three alternatives for keeping the index updated. As we will see in Section 2.6, each of these gives very poor performance. The poor performance of the traditional approach of building an index on the data (i.e., the objects) can be traced to the following two problems: i) whenever *any* object moves, it becomes necessary to re-execute *all* queries; and ii) the cost of keeping the index

updated is very high. In the next two sections we develop two novel indexing schemes that overcome these limitations.

2.4 Query indexing: Queries as data

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

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

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

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

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

is accessed. For objects that move, the Q-index improves the search performance as compared to a comparison against all queries.

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

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

2.4.1 *Safe region*s: Exploiting query and object locations

Consider an object that is far away from any query. This object has to move a large distance before its relevance to any query changes. Let *SafeDist* be the shortest distance between object O and a query boundary. Clearly, O has to move a distance of at least *SafeDist* before its relevance with respect to any query changes. Thus we need not check the Q-index with O's new location as long as it has not moved by *SafeDist*.

Note that when we talk about *SafeDist* not as distance but as an *optimization*, we mean that each object tracks the distance it traveled, not the shortest straight line distance from the object current location to the last recorded location, see Section 2.6.2 for more details.

Similarly, we can define two other measures of "safe" movement for each object:

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

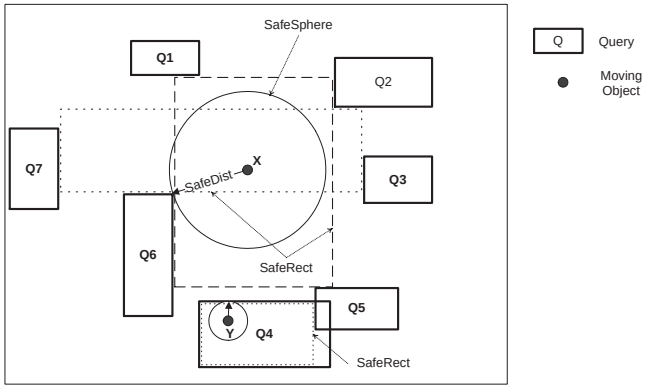


Figure 2.2 Examples of *Safe Regions*

Figure 2.2 shows examples of each type of *Safe Region*. Note that it is not important whether an object lies within or outside a query that contributes to its safe region. Points X and Y are examples of each type of point: X is not contained within any query, whereas Y is contained in query Q_1. The two circles centered at X and Y are the *SafeSphere* regions for X and Y respectively, and the radiuses of the two circles are their corresponding *SafeDist* values. Two examples of *SafeRect* are shown for X. The *SafeRect* for Y is within Q_4. Note that for X, other possibilities for *SafeRect* are possible. With each approach, only objects that move out of their safe region need to be evaluated against the Q-index. These measures identify ranges of movement for which an object's matching does not change and thus it need not be checked against the Q-index. This significantly reduces the number of accesses to Q-index. Note that for the *SafeDist* technique, we need to keep track of the total distance traveled since *SafeDist* was computed. Once an object has traveled more than *SafeDist*, it needs to be evaluated against the Q-index until *SafeDist* is recomputed. On the other hand,

for the *SafeSphere* and *SafeRect* measures, an object could exit the safe region, and then re-enter it at a later time. While the object is inside the safe region it need not be evaluated against Q-index. While it is outside the safe region, it must be evaluated at each time step.

The safe region optimizations significantly reduce the need to test data points for relevance to queries if they are far from any query boundaries and move slowly, see Section 2.6.2 for the experimental results. Recall that each object reports its location periodically or when it has moved by a significant distance since its last update. This decision can be based upon safe region information sent to each object. Thus the object need not report its position when it is within the safe region, thereby reducing communication and the need for processing at the server. The effectiveness of these techniques in reducing the number of objects that need to report their movement is studied in Section 2.6. Even though we do not perform any re-computation of the safe regions in our experiments, we find that the safe region optimizations are very effective. It should be noted that multiple safe regions can be combined to produce even larger safe regions. By definition, there are no query boundaries in a safe region. Hence there can be no query boundary in the union of the two safe regions.

2.4.2 Computing the safe regions

The Q-index can be used to efficiently compute each of the safe regions. *SafeDist* is closely related to a nearest-neighbor query since it is the distance to the nearest query boundary. A branch-and-bound algorithm similar to that proposed for nearest neighbor queries in [RKV95] is used. The [RKV95] algorithm prunes the search based upon the distances to queries and bounding boxes that have already been visited. Our *SafeDist* algorithm is different in that the distance between an object and a query is always the shortest distance from the object to a boundary of the query. In [RKV95] this distance is zero if the object is contained within the query[2]. To amortize the cost of *SafeDist* computation, we combine it with the evaluation of the object on

[2]Please note that in [RKV95] the role of objects and queries is not reversed as it is here.

the Q-index, i.e., we execute a combined range (objects are tiny rectangles) and a modified nearest-neighbor query. The modification is that the distance between an object and a query is taken to be the shortest distance to any boundary even if the object is contained in the query (normally this distance is taken to be zero for nearest-neighbor queries). The combined search executes both queries in parallel thereby avoiding repeated retrieval of the same nodes. *SafeSphere* is simply a circle centered at the current location of the object with a radius equal to *SafeDist*.

Given an object and a set of query rectangles, there exist various methods for determining safe rectangles. The related problem of finding a largest empty rectangle has been studied extensively and solutions vary from $O(n)$ to $O(n \log^3 n)$ time, (where n is the number of query rectangles) depending on restrictions on the regions [AS87, AW88, Ame94, MOS85]. For our application, finding the "best", or maximal rectangle is not important for correctness (any empty rectangle is useful), we use a simple $O(n^2)$ time implementation for computing a safe rectangle. The implementation allows adaptations leading to approximations for the largest empty rectangle. The algorithm for finding the *SafeRect* for object O is as follows:

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

2. Take object O as the origin and determine which relevant rectangles lie in which of the four induced quadrants. For each quadrant, sort the corner vertices of query rectangles that fall into this quadrant. For each quadrant determine the dominating points [CLR90].

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

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

2.5 Velocity constrained indexing

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

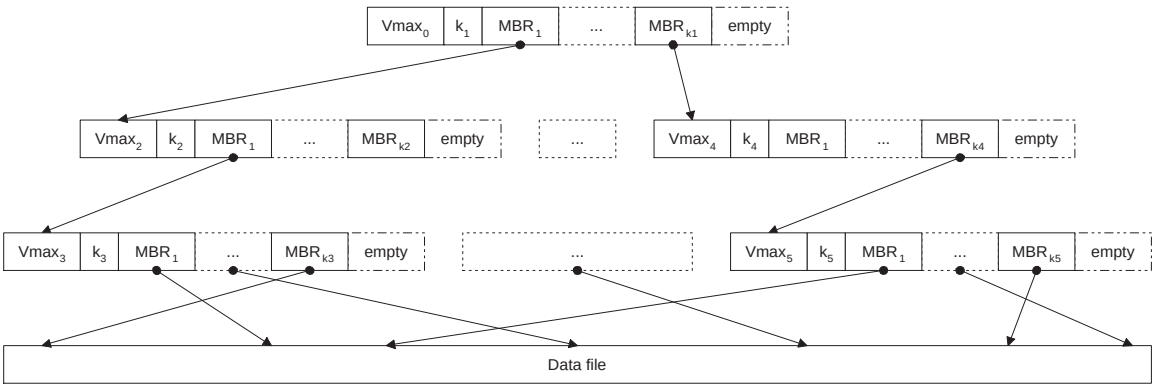


Figure 2.3 Example of Velocity Constrained Index (VCI)

A VCI is a regular R-tree based index on moving objects with an additional field in each node: v_{max}. This field stores the maximum allowed speed over all objects

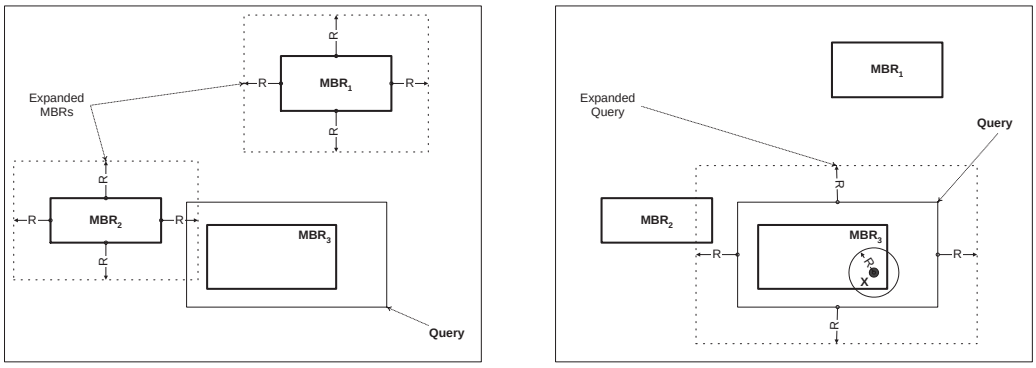
covered by that node in the index. The v_{max} entry for an internal node is simply the maximum of the v_{max} entries of its children. The v_{max} entry for a leaf node is the maximum allowed speed among the objects pointed to by the node. Figure 2.3 shows an example of a VCI. The v_{max} entry in each node is maintained in a manner similar to the MBRs of each entry in the node, except that there is only one v_{max} entry per node as compared to an MBR per entry of the node. When a node is split, the v_{max} for each of the new nodes is copied from the original node.

Consider a VCI that is constructed at time t_0. At this time it accurately reflects the locations of all objects. At a later time t, the same index does not accurately capture the correct locations of points since they may have moved arbitrarily. Normally the index needs to be updated to be useful. However, the v_{max} fields enable us to use this old index without updating it. We can safely assert that no point will have moved by a distance larger than $R = v_{max}(t - t_0)$. If we expand each MBR by this amount in all directions, the expanded MBRs will correctly enclose all underlying objects. Therefore, in order to process a query at time t, we can use the VCI created at time t_0 without being updated, by simply comparing the query with expanded version of the MBRs saved in VCI. At the leaf level, each point object is replaced by a square region of side $2R$ for comparison with the query rectangle[3].

An example of the use of the VCI is shown in Figure 2.4(a) which shows how each of the MBRs in the same index node are expanded and compared with the query. The expanded MBR captures the worst-case possibility that an object that was at the boundary of the MBR at t_0 has moved out of the MBR region by the largest possible distance. Since we are storing a single v_{max} value for all entries in the node, we expand each MBR by the same distance, $R = v_{max}(t - t_0)$. If the expanded MBR intersects with the query, the corresponding child is searched. Thus to process a node we need to expand all the MBRs stored in the node (except those that intersect without expansion, e.g. MBR_3 in Figure 2.4). Alternatively, we could perform a single expansion of the query by R and compare it with the unexpanded MBRs.

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

An MBR will intersect with the expanded query if and only if the same MBR after expansion intersects with the original query. This is because range queries and MBRs are always iso-oriented, and therefore the sides are always parallel to the coordinate axes. Figure 2.4 (b) shows the earlier example with query expansion. Expanding the query once per node saves some unnecessary computation.



(a) (b)

Figure 2.4 Query Processing with Velocity Constrained Index (VCI)

The set of objects found to be in the range of the query based upon an old VCI is a superset, S' of the exact set of objects that currently are in the query's range. Clearly, there can be no false dismissals in this approach. In order to eliminate the false positives, it is necessary to determine the current positions of all objects in S'. This can be achieved through a post-processing step. The current location of the object is retrieved and compared with the query to determine the current matching.

Note that it is not always necessary to determine the current location of each object that falls within the expanded query. From the position recorded in the leaf entry for an object, it can move by at most R. Thus its current location may be anywhere within a circle of radius R centered at the position recorded in the leaf. If this circle is entirely contained within the unexpanded query, there is no need to post-process this object for that query. Object X in Figure 2.4(b) is an example of such a point.

It should be noted that although the expansion of MBRs in VCI and the time-evolving MBRs proposed in [SJLL00] are similar techniques, the two are quite different in terms of indexing of moving objects. A key difference between the two is the model of object movement. Saltenis et al. [SJLL00] assume that objects report their movement in terms of velocities (i.e., an object will move with *fixed speed* in a *fixed direction* for a period of time). In our model the only assumption is that an object cannot travel faster than a certain known velocity. In fact, for our model the actual movement of objects is unimportant (as long as the maximum velocity is not exceeded). The time varying MBRs [SJLL00] exactly enclose the points as they move, whereas VCI pessimistically enlarges the MBRs to guarantee enclosure of the underlying points. Thus VCI requires no updates to the index as objects move, but post-processing is necessary to take into account actual object movement. The actual movement of objects has no impact on VCI or the cost of post-processing. Of course, as time passes, the amount of expansion increases and more post-processing is required.

2.5.1 Optimizations of VCI

To avoid performing an I/O operation for each object that matches each expanded query, it is important to handle the post-processing carefully.

Optimization1 We can begin by first pre-processing all the queries on the index to identify the set of objects that need to be retrieved for any query. These objects are then retrieved only once and checked against all queries. This eliminates the need to retrieve the same object more than once.

Optimization2 We could still retrieve the same page containing several objects multiple times. To avoid multiple retrievals of a page, the objects to be retrieved are sorted on page number.

Optimization3 We can build a clustered index. Clustering may reduce the total number of pages to be retrieved. When clustering is used the order of objects in the file storing their locations is organized according to the order of entries in the leaves

of the VCI. Clustering can be achieved efficiently following creation of the index. A depth first traversal of the index is made and each object is copied from the original location file to a new file in the sequential order and the index pointer is appropriately adjusted to point to the newly created file. By default the index is not clustered. As is seen in Section 2.6, clustering the index improves the performance by roughly a factor of 3.

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

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

followed by a less frequent rebuild. Experimentally, we found that refreshing works very well.

2.6 Experimental evaluation

In this section we present the performance of the new indexing techniques and compare them to existing techniques. The experiments reported are for two-dimensional data, however, the techniques are not limited to two dimensions. The various indexing techniques were implemented as R^*-trees [BKSS90] and tested on synthetic data. The dataset used consists of 100,000 objects composed of a collection of 5 normal distributions each with 20,000 objects. The mean values for the normal distribution are uniformly distributed, and the standard deviation is 0.05 (the points are all in the unit square). The centers of queries are also assumed to follow the same distribution but with a standard deviation of 0.1 or 1.0. The total number of queries is varied between 1 and 10,000 in our experimentation. Each query is a square of side 0.01. Other experiments with different query sizes were also conducted but since the results are found to be insensitive to the query size, they are not presented. More important than query size is the total number of objects that are covered by the queries and the number of queries.

The maximum velocities of objects follow a Zipf distribution with an overall maximum value of V_{max}. For most experiments V_{max} was set to 0.00007 – if we assume that the data space represents a square of size 1000 miles (as in [KGT99]), this corresponds to an overall maximum velocity of 250 miles an hour. In each experiment, we fix the fraction of objects, m, that move at each time step. This parameter was varied between 1000 and 10,000. The time step is taken to be 50 seconds. At each time step, we randomly select m objects and for each object, we move it with a velocity between 0 and the maximum velocity of the object in a random direction. The page size was set to 2048 bytes for each experiment. As is customary, we use the number of I/O requests as a metric for performance. The top two levels of each index were

assumed to be in memory and are not counted towards the I/O cost. The various parameters used are summarized in Table 2.1.

Table 2.1 Parameters used in the experiments

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

2.6.1 Traditional schemes

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

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

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

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

The *modify* approach is similar to the technique for handling movement of points proposed by Saltenis et. al. [SJLL00] wherein the bounding boxes of the nodes are expanded to accommodate the past, current, and possibly future positions of objects. The *modify* approach differs from these because it does not save past or future positions in the index which is acceptable since the purpose of this index is primarily to answer continuous queries based upon the current locations of the objects. The approach of [SJLL00] assumes that objects move in a straight line with a fixed speed most of the time. Whenever the object's speed or velocity changes, an update

is sent. The index is built using this speed information. Their experimental results are based upon objects moving between cities which are assumed to be connected by straight roads and the objects moves with a very regular behavior – for the first sixth of a route they accelerate at a constant rate to one of three maximum velocities which they maintain until the last sixth of the route at which point they decelerate. Our model for object movement is more general and does not require that object maintain a given velocity for any point in time. Under this model, the approach of [SJLL00] is not applicable.

Table 2.2 shows the relative performance of these schemes in terms of number of I/O operations performed. The performance of these approaches does not vary over time, hence we simply report a single value for each combination of m and q. We assume that the top two levels of the index are memory resident. For these experiments that roughly corresponds to about 21 pages in memory. The I/O numbers for *Brute Force* are evaluated assuming efficient use of 21 buffers: 20 buffers are assumed to hold blocks of queries.

Table 2.2 Performance of traditional techniques.

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

From the table it is clear that the *Brute Force* approach gives the best performance in all cases. This is largely due to the fact that this approach does not need to maintain any structures as objects move. We assume that there are enough buffers to hold only the first two levels of the other indexes when computing brute force. The *Reconstruct*

approach is clearly the poorest since it is too expensive to build the index at each time step. The *Insert/Delete* scheme incurs roughly double the I/O cost that *Modify* incurs to update the index while their query cost is the same.

We point out that while the *Brute Force* approach has the lowest I/O cost, it may not be the best choice. The reason is that unlike the other schemes which employ an index on the objects to significantly reduce the number of comparisons needed, *Brute Force* must compare each object with each query. Thus it is typically going to incur almost three orders of magnitude more comparisons than the others! An earlier experiment to measure the total time required for *Modify* and *Brute Force* showed that their performance is very comparable despite the low I/O cost of *Brute Force* [AHP00]. Except for this special case, the I/O cost is a good measure of performance.

2.6.2 *Safe region* optimizations

Since the Query-indexing is a joint work with several people, here you can find only a summary of the performance of the query indexing approach. Please refer to [PXK+02] for figures and details on the Query-indexing.

We now study the performance of the safe region optimizations among themselves. For each scheme we study the *Reduction Rate*: the fraction of moved objects that are within their safe region. These objects do not need to report their location. We study the effectiveness of each measure as time passes. We evaluated various combinations of m and q. For example, we investigated the reduction rates for 10% objects moving at each time step, and 1000 queries. Each of the safe regions is computed at time 0. As long as the object is within its safe region it does not report its new location. As expected, the fraction of objects that remain within their safe regions drops as time passes. For example after 100 seconds 95% of the objects are still within their *SafeRect* and need not report their positions, whereas 83% of the objects are within their *SafeDist*. Thus *SafeRect* is more effective in reducing the need for objects reporting their locations. An important point to note is that even though we do not re-compute the safe regions in our experiments, we find that the safe region optimizations remain

very effective for large durations. This is important since the cost of computing these measures is high. The cost of computing *SafeDist* and *SafeSphere* is on the order of 13 I/O operations per object for the case of 10,000 queries. The cost of computing *SafeRect* is significantly higher – around 52 I/Os per object. These high costs do not adversely affect the gains from these optimizations since the re-computations are done infrequently.

A common trend is that the *SafeRect* measure is most effective. *SafeSphere* is never worse in performance than *SafeDist*. This is not surprising since *SafeSphere* augments *SafeDist* with the center information to provide a safe region as opposed to an absolute measure of movement. Thus under *SafeSphere* an object can re-enter the safe region whereas an object that has moved by *SafeDist* must always be tested against Q-index until *SafeDist* is re-computed. To see why *SafeRect* outperforms *SafeSphere*, consider that the *SafeSphere* pessimistically limits the region of safety in all directions by the shortest distance from the object to a query boundary. On the other hand, *SafeRect* selects four distances, one in each direction, to the nearest query boundary. Thus it is likely to extend further than the *SafeSphere* [4]. This is especially the case when the number of queries gets very large. The optimizations remain effective even when the number of moving objects is increased ten-fold to 10,000.

2.6.3 Incremental evaluation and the Q-index approach

We compared the performance of the Q-index approach with the traditional approaches. Let us first assume that there is enough memory available to keep the Q-index entirely memory resident. This assumption can be done because the number of queries typically is much smaller than the number of moving objects and correspondingly an index on queries can fit in main memory. Under this assumption at each time step, we simply need to read in the positions of only those objects that

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

have moved since the previous evaluation. A separate file containing only these points can be easily generated on the server during the period between evaluations. Thus the I/O cost of the Q-index approach is simply given by the number of pages that make up this file. The safe region optimizations further reduce the need for I/O by effectively reducing the number of objects that report their updates. We investigated the performance of the various schemes.

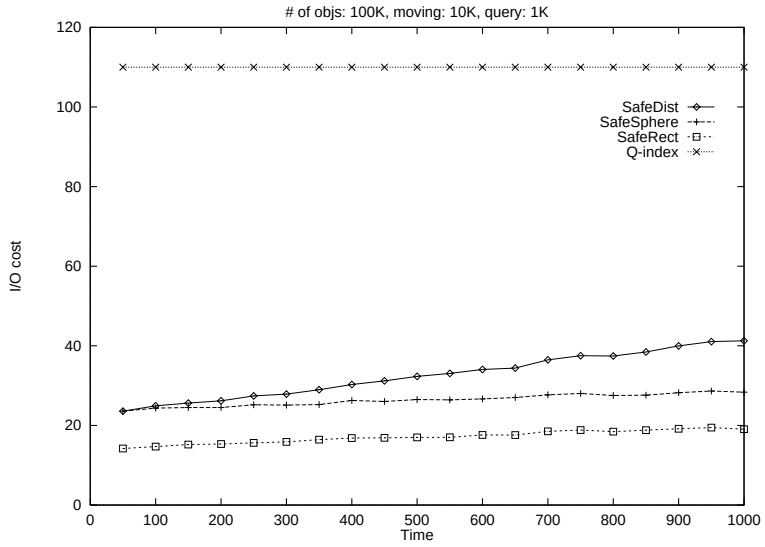


Figure 2.5 Performance of the Q-index technique with 10% moving and 1% queries

In Figure 2.5 the results with 1000 queries and 10,000 objects moving at each time step are shown. The Q-index approach requires 110 pages of I/O at each time step to retrieve the new locations of the moved objects and process them against the memory-resident Q-index. It should be pointed out that this is actually sequential I/O. This is a significant reduction from the I/O cost of the traditional approaches as shown in Table 2.2 representing more than an order of magnitude improvement. The optimizations further reduce the I/O cost by almost another order of magnitude. Of course, this reduction reduces over time but not significantly even for as many as 1000 seconds.

For smaller numbers of moving objects, the Q-index needs to perform proportionately smaller numbers of I/O operations. If only 1% of objects is moving than only 20 I/Os are needed at each time step for Q-index. As the number of objects that move at each time step is increased, the Q-index approach needs to perform increased I/O until eventually a sequential scan of the entire data file is required when virtually every object moves in each time step. Thus the approach scales well with the movement of objects, gracefully degrading to a sequential scan.

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

If we consider only I/O for *Brute Force*, an incremental version can outperform the Q-index based approach if the Q-index does not fit in memory. Consider the above case with 10,000 queries and 1000 objects moving at each time step. This corresponds to 100 pages for queries and 10 pages for objects that move at each time. If we assume that $B+1$ buffers are available, then *brute force* can evaluate all queries with $\lceil \frac{100}{B} \rceil (B + 10)$ I/O operations. With as few as 21 buffers, this requires only 150 I/Os as compared to over 1800 for Q-index! However, as pointed out earlier, the *Brute Force* approach pays a high computation price that offsets the reduced I/O. To validate this claim, we conducted an experiment where we measured the total time taken (in seconds) by *Brute Force* and the other proposed approaches. The results

are shown in Table 2.3 for two sets of values of m and q. We see that although *Brute Force* would have only a fraction of the I/O operations required by the others, it is actually slower overall. We again point out that this anomaly of I/O time not translating to overall performing happens only for *Brute Force* due to its inordinately large numbers of comparisons.

Table 2.3 Impact of CPU cost

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

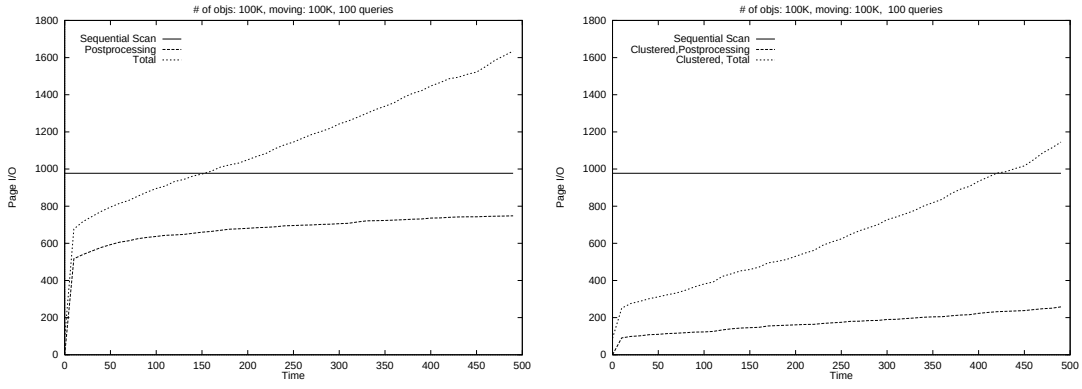
Impact of Velocity. From the above experiments we find that the incremental Q-index approach and optimizations scale well with variations in the number of moving objects and queries. To studied the impact of the degree of movement of the objects, we conducted experiments where we altered the speed distributions. The results with a V_{max} corresponding to 500 miles per hour (not shown) are very similar to that of 250mph except that the optimizations are a little less effective. The relative performance of the newly proposed schemes is unaffected by these changes in the allowable speeds of objects. We see only a slight change in the effectiveness of the optimizations.

Denser Object Locations. We investigated the effect of a much denser set of objects and queries corresponding to a smaller region such as a city. In our experiment the range of the space is reduced from 1000 miles by 1000 miles to a 10 mile by 10 mile region. The number of objects is maintained at 100,000 with 1000 moving at each time step. The number of queries is 10,000. Since a city is likely to have a more uniform distribution, we increased the standard deviation to 0.8. Also the maximum speed is reduced from 250 miles per hour to 50 miles per hour. In comparison with the wider area setting we can see that the performance is poorer by about a factor of

10. This reduction in performance is not surprising since the safe region optimizations are less effective. This is directly related to the rate at which objects exit their safe regions. The important point however, is that the Q-index approaches are still an order of magnitude better than the traditional approaches.

2.6.4 Velocity constrained indexing

Next we discuss the performance of the Velocity Constrained Indexing (VCI) algorithm. There are two components of the cost for VCI: i) pre-processing to evaluate the expanded queries on VCI; and ii) post-processing to eliminate false positives. Since the VCI approach is unaffected by the actual number of objects that move at each time step (i.e., m), all objects were moved at each time step. Figure 2.6(a) shows the performance of VCI for 100,000 objects moving at each time instant, and 100 queries. The pre-processing cost increases with time since the queries get larger due to greater expansion resulting in more parallel path searches on the VCI. Similarly, post-processing cost increases with time since more and more false positives are likely to be found with increased query expansions. The graph shows the post-processing cost and the total cost as time since creation of VCI. The cost of a sequential scan of the entire object file is also shown.



(a) (b)

Figure 2.6 Performance of Velocity Constrained Indexing (a) No Clustering; (b) With Clustering

The total cost approaches that of a sequential scan after about 150 seconds, at which point the VCI is not effective – it would be more efficient to scan the file instead of using the index. Figure 2.6(b) shows the improvement due to clustering. There is a very significant improvement in post-processing cost resulting in about 400 fewer I/O operation at each time step. Thus clustering extends the utility of the VCI from about 150 seconds to over 400 seconds.

Refresh and Rebuild. Figure 2.7 shows the impact of applying a *refresh* to the VCI. The refresh helps reduce both the pre- and post-processing costs. The pre-processing cost is reduced since the MBRs better fit the underlying data and the clock for query expansion is reset. This improves the quality of the index resulting in faster query processing. The post-processing cost is reduced since there will be fewer false positives as a result of a "tighter" index. The overall cost is reduced by almost 600 I/O operations immediately following the refresh. Over time it again degrades and another refresh is applied, etc. The refresh period can be adjusted as necessary. In this experiment, the refresh was performed to keep the total cost below that of a sequential scan. The application of a rebuild has a very similar effect to that of a refresh. The difference would show up only for very large time intervals when objects have moved so much that the old VCI organization is inefficient. The effect of rebuilding would be very similar to that of running the test again from second 0, hence we do not consider it here.

Sensitivity to Parameters. The VCI approach is not affected by the actual movement of objects (other than through the maximum speeds). Thus the costs would not change even if all the objects were moving at each time instant! On the other hand, the VCI approach is very sensitive to the number of queries. The above graphs are for only 100 queries. If the number of queries is increased to 1000, we find that the pre-processing cost increases proportionately, as does the post-processing cost. Very soon after creation of the VCI its performance degrades to worse than a sequential scan forcing frequent refreshing. This impacts the performance and renders the scheme unusable. Thus VCI is a reasonable approach when the queries

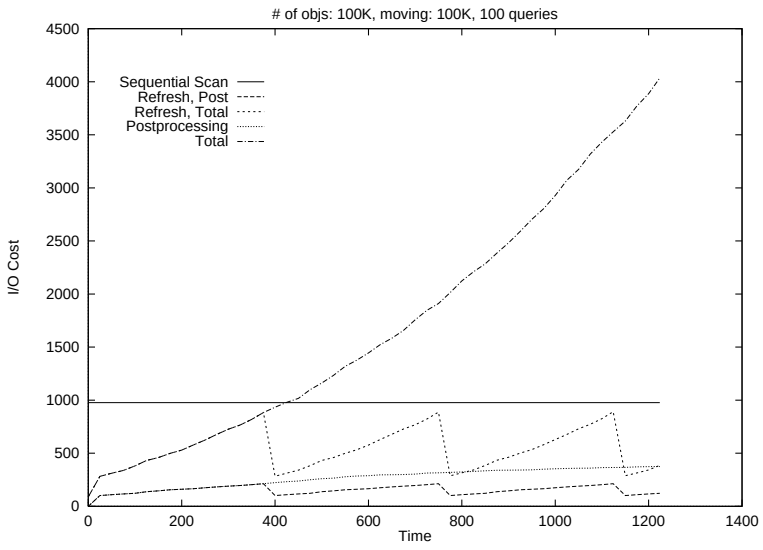


Figure 2.7 Impact of Refresh on Velocity Constrained Indexing

cover a small number of objects. With 10 queries, we find that the graph scales down roughly linearly too, e.g. for a single query the post-processing and total costs are 6, and 17 I/Os, respectively. To study the impact of denser distributions of queries, we repeated the above tests for the VCI approach with a query set having one-tenth the standard deviation of the other tests (viz. 0.1). The results are shown in Figure 2.8. The relative performance of the graphs is very similar to that seen with a broader distribution, except that the degradation towards a sequential scan occurs much faster. This is expected since each query now covers more objects on average.

The maximum speed of objects is clearly an important parameter for the VCI approach. To study the impact of V_{max} (the overall maximum speed of any object) on performance we conducted several experiments with different values of V_{max}. The results resemble very closely those obtained earlier for VCI. The major impact of changes in maximum speed is that the time scale get stretched (contracted) if V_{max} is reduced (increased). The stretching is linearly related to the changes in speed. Other than this difference, the graphs are very similar. This is not surprising because the important factor in determining the performance of VCI is the amount of expansion that a query experiences: $R = v_{max}(t - t_0)$ (v_{max} is the maximum velocity field

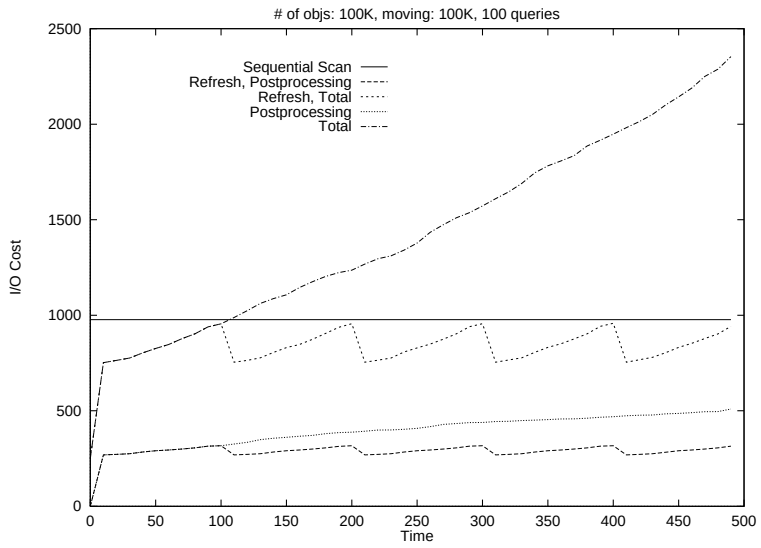


Figure 2.8 Performance of Velocity Constrained Indexing with query std = 0.1

stored in the node being examined). With double the speed, we need half the time difference to achieve the same expansion. Therefore if the max speed in increased by a large factor, such as with the experiment on the 10 miles by 10 miles range (this is effectively increasing the speed of the objects) VCI becomes ineffective. With 10 queries, a sequential scan would be better after only 45 seconds.

Figures 2.9 and 2.10 demonstrate the I/O cost of the brute force approach and VCI approach when the cost of the refresh procedure itself is included. The time step in both experiments is assumed to be 25 seconds, which is a very large time step. Large time step gives advantage to the brute force method, since with smaller step the cost of refresh procedure is better amortized. Even with such a large time step the VCI technique with the refresh approach significantly outperforms the brute force method. If data do not preserve locality, it is expected that after substantial amount of time the cost of the VCI with the refresh will degrade and the index will need to be rebuilt.

The problem that is yet to be addressed is the question of how to pick appropriate period between two successive refreshes to maximize performance. The refreshes cannot be carried out out too often because of the cost of the refresh procedure,

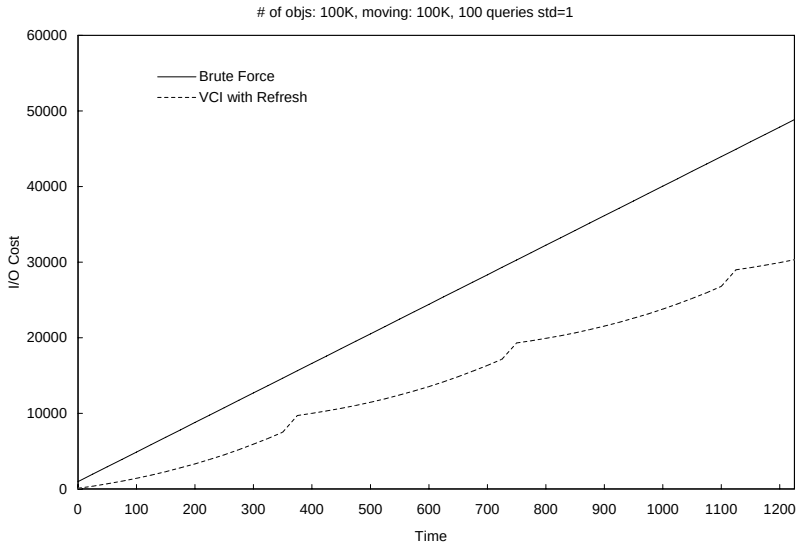


Figure 2.9 Cumulative I/O Cost, cost of Refresh procedure is included, time step 25 secs

and they cannot be carried out infrequently because the performance will eventually degrade significantly. That is why there is the optimal inter-refresh period, which need to be found. We plan to address this as future work.

Comparison to Q-index. Our experimental work indicates that the Q-index approach outperforms the VCI approach. For even a hundred queries, VCI incurs between 280 and 880 I/O operations (Figure 2.6). For larger numbers of queries, it will certainly not incur any less since each extra query will add to the query processing cost as well as potentially generate new objects that need to be post-processed. In contrast, for 1000 queries, Q-index needs 110 I/Os without safe region optimizations, and only about 20 I/Os with the *SafeRect* optimization. A positive aspect of VCI is that it is insensitive to variations in the number of moving objects, m. Even if all the objects move, the Q-index approach[5] will incur a sequential scan. Thus it is possible that for very few queries and very large numbers of objects moving at each time instant, VCI could outperform Q-index, however this is not very practical.

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

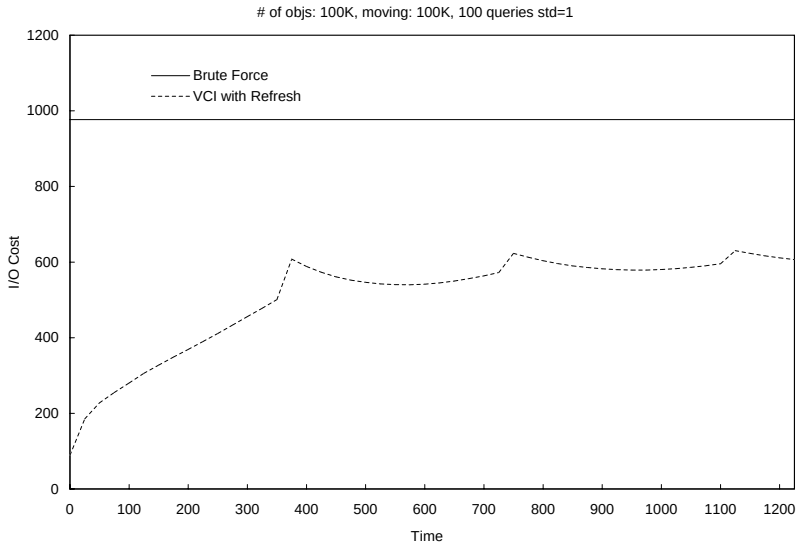


Figure 2.10 Average (per time step) I/O Cost, cost of Refresh procedure is included, time step 25 secs

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

2.6.5 Combined indexing scheme

The results show that query indexing and safe region optimizations significantly outperform the traditional indexing approaches and also the VCI approach. These improvements in performance are achieved by eliminating the need to evaluate all objects at each time step through incremental evaluation. Thus they perform well when there is little change in the queries being evaluated. The deletion of queries can be easily handled simply by ignoring the deletion until the query can be removed from the Q-index. The deleted query may be unnecessarily reducing the safe region for some objects, but this does not lead to incorrect processing and the correct safe

regions can be recomputed in a lazy manner without a significant impact on the overall costs.

The arrival of new queries, however, is expensive under the query indexing approach as each new query must initially be compared to every object. Therefore a sequential scan of the entire object file is needed at each time step that a new query is received. Furthermore, a new query potentially invalidates the safe regions rendering the optimizations ineffective until the safe regions are recomputed. The VCI approach, on the other hand, is unaffected by the arrival of new queries (only the total number of queries being processed through VCI is important). Therefore to achieve scalability under the insertion and deletion of queries we propose a *combined scheme*. Under this scheme, both a Q-index and a Velocity Constrained Index are maintained. Continuous queries are evaluated incrementally using the Q-index and the *SafeRect* optimization. The Velocity Constrained Index is periodically refreshed, and less periodically rebuilt (e.g., when the refresh is ineffective in reducing the cost). New queries are processed using the VCI. At an appropriate time (e.g., when the number of queries being handled by VCI becomes large) all the queries being processed through VCI are transferred to the Query Index in a single step. As long as not too many new queries arrive at a given time (e.g., less than 10 in each time step), this solution offers scalable performance that is orders of magnitude better than the traditional approaches.

We now present the performance of the combined scheme. The experiment is conducted with 100,000 objects with 1% moving. The experiment begins with 10,000 queries and new queries arrive at the rate of 10 queries every three minutes (actually one query every 18 seconds). Newly arriving queries are handled by the VCI index while the ongoing queries are processed using the Q-index. When the number of queries handled by the VCI index reaches a threshold (100 in this experiment), we ingest the 100 queries into the Q-index in a single step. This ingestion requires changes to the index structure and also potentially changes the safe regions for the objects. We consider two approaches for correcting the safe regions: recomputing

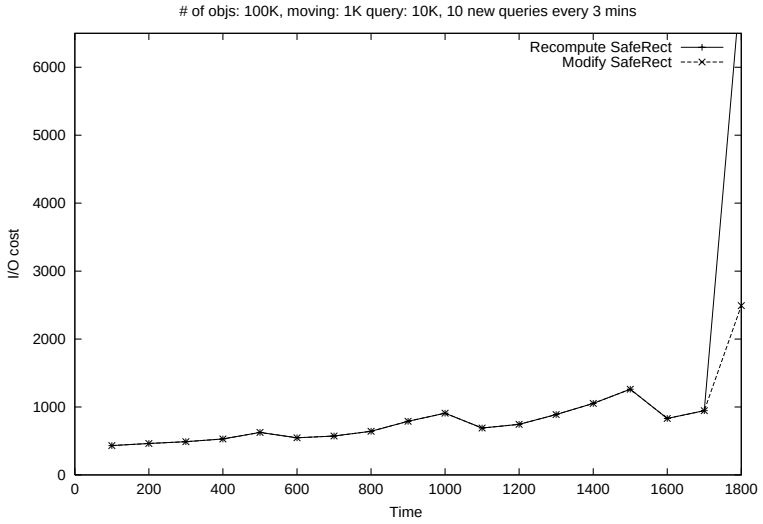


Figure 2.11 Performance of VCI and Q-index With Dynamic Queries

all the safe regions, and modifying the safe regions by comparing them against the ingested queries. Figure 2.11 shows the combined cost of the two indexes over time as objects arrive. Only the SafeRect region is considered. As can be seen from the graph, the combined cost remains very small until the point at which the queries are ingested. At this time, a large penalty is paid for computing the new SafeRect regions. The recompute approach is very expensive, however the modify approach does not incur a very large overhead. The effect of three refreshes on the VCI is clearly visible.

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

2.7 Conclusion

Moving object environments are characterized by large numbers of moving objects and concurrent active queries over these objects. Efficient continuous evaluation of these queries in response to the movement of the objects is critical for supporting acceptable response times. We showed that the traditional approach of building an index on the objects (data) can result in poor performance. In fact, a brute force, no index strategy gives better performance in many cases. Neither the traditional approach, nor the brute force strategy achieve reasonable performance.

We presented a novel indexing technique for scalable execution called *Velocity Constrained Indexing (VCI)* and compared it to the *Query Indexing* approach. We showed that the VCI approach gives good performance for hundreds of concurrent queries, and, unlike the query indexing approach, it is unaffected by changes in queries and actual object movement. We showed that the VCI and Query-indexing techniques complement each other enabling a combined solution that efficiently handles not only ongoing queries but also dynamically inserted queries. The experiments also demonstrated the robustness of the new techniques to variations in the parameters. The combined schemes therefore achieve superior performance to existing solutions for the efficient and scalable evaluation of continuous queries over moving objects.

3. IN-MEMORY QUERY INDEXING

3.1 Introduction

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 makes main memory evaluation highly attractive. The growing importance of main memory based algorithms has been underscored by the Asilomar report [Ber98], 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 [KCK01], 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 chapter we investigate several in-memory index structures for efficient and scalable processing of continuous queries. Towards the goal of wider applicability of our algorithms, we make no assumptions neither about the speed and nature of the movement of objects nor about 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. 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. For example, 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 chapter is organized as follows. Section 3.2 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 3.3 presents the experimental results. Section 3.4 concludes the chapter.

3.2 Continuous query evaluation

3.2.1 Model of object movement

The model of object movement is similar to that of Section 2.3.3, but with a few specifics for main memory. 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 that was not discussed in Section 2.3.3.

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., [SWCD97]), *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.).

3.2.2 Query indexing

This chapter builds on the idea of *Query Indexing* proposed in Section 2.4. 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).

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 point to query mapping, handle region queries, answer simple density queries (e.g. monitor how many people are in a building) etc.

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.

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 3.1 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 points. 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 point 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 points 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 3.1. 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. 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 3.3 we evaluate the performance of these alternative indexes.

3.2.3 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 [KCK01], on the other hand, is a variant of R-trees optimized for main memory. All of these indexes were implemented for main memory use. 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 [KCK01]. The main idea is to make the R-tree cache-conscious by compressing MBRs. The R-tree is a balanced tree where each node has so called Minimum Bounding Rectangles (MBRs) associated with it.

All object that a node covers are tightly bounded by the node's MBR. Each node also contains MBRs of its children. Processing of various types of queries normally starts from the root node and proceeds based on the MBR information in each considered node. The R*-tree is an R-tree which utilizes different heuristics for splitting a node when it is overfull.

The CR-tree proposes a way for compressing MBRs by using so-called Quantized Relative Minimum Bounding Rectangles (QRMBR). Other well-known minor optimizations have also been proposed 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 3.2 shows an example of a grid. Throughout the chapter, we assume that the

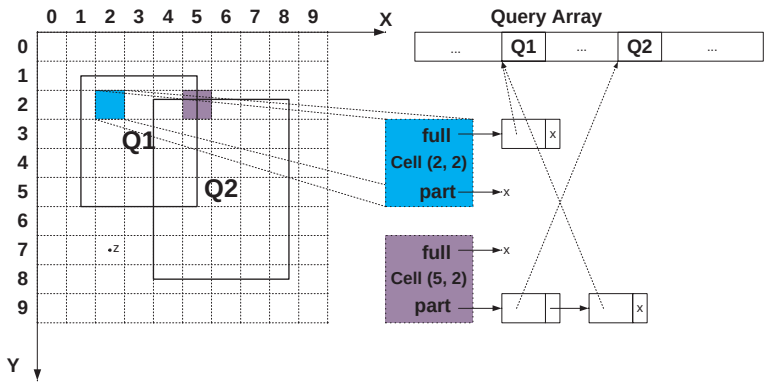


Figure 3.2 Example of grid

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 3.2). 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[1] rather than lists, improving performance by roughly 40% due to the achieved clustering. An analytical solution for the appropriate choice

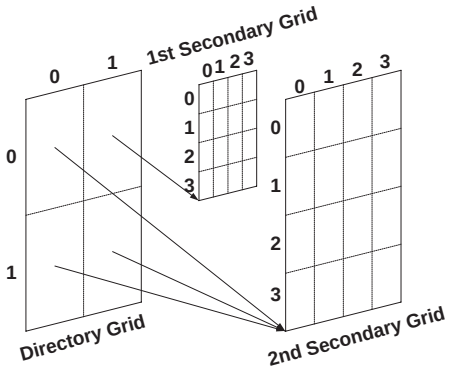


Figure 3.3 Example of grid with two tiers

of grid size is presented in Section 3.2.4. 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 3.3. 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 is unlikely to lead to a 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

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

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" [KPAH02].

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. 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. 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.

3.2.4 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

Table 3.1 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	$x = q - i \times c$

queries with side q, uniformly distributed on $[0,1]^2$ domain, are added to index, see Figure 3.4a. Let c be the side of each cell. Then query side q can be presented as

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

Cell $G(0,0)$ can be logically divided into three parts (or sets), see Figure 3.4a. *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 3.4b), for

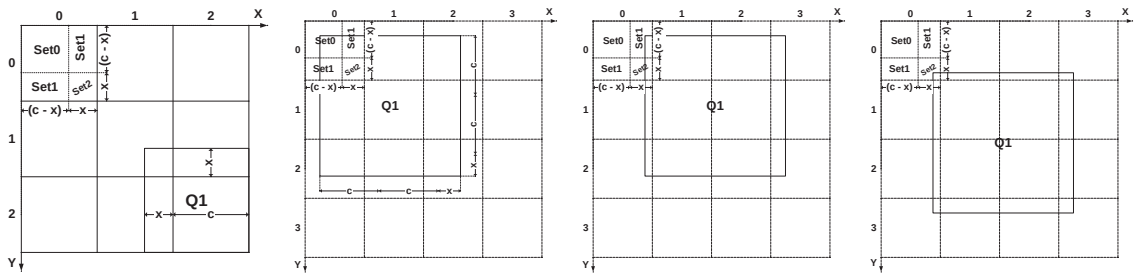


Figure 3.4 Example of query (a) choosing grid size (b) top-left corner in *Set0* (c) top-left corner in *Set1* (d) top-left corner in *Set2*

Set1 this number is $4i+2$ (see Figure 3.4c), and for *Set2* it is $4i+4$ (see Figure 3.4d). 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 *Set2* is $\frac{x^2}{c^2}$. Therefore, on 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 a 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 Figure 3.4. 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 3.13a. 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.

3.2.5 Memory requirements for grid

Let us first compute average size of *full* list of each cell in an analogous fashion as in Section 3.2.4.

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 $(i-1)^2$ part lists (see Figure 3.4b), for *Set1* this number is $i(i-1)$ (see Figure 3.4c), and for *Set2* it is i^2 (see Figure 3.4d). 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 *Set2* is $\frac{x^2}{c^2}$. Therefore, on average, each query ends up in avg number of part lists:

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

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

Case 2: Query side is less than cell side: $q < c$. In this case the size of each *full* list is zero.

Final formula The space that a grid occupies in main memory can be estimated as the sum of the space that is occupied be cells without the lists and of the space occupied by the lists. The former can be computed as the number of cells times the cell size, i.e. $S_{cells} = \frac{1}{c^2} 2S_p$, where S_p is the size needed in the system to store a pointers (e.g. 4). This is so because each cell simply contains two pointers to its *full* and *part* lists. The space occupied by the lists S_{lists} can be computed as the number of cells times the space needed for *full* and *part* lists per cell on average. Let us first compute the average combined size N_{list} of *full* and *part* lists per cell:

$$N_{list} = \begin{cases} 4qmc + m(q-c)^2 = m(q+c)^2 & \text{if } q > c; \\ m(q+c)^2 & \text{otherwise.} \end{cases}$$

$$N_{list} = m(q+c)^2$$

Since there are c^{-2} cells, the space occupied by all lists can be computed as $N_{lists} = \frac{N_{list}}{c^2}$. Each element of the list consists of a pointer to a query and a pointer to the next element, therefore $S_{lists} = N_{lists} \times 2S_p$.

Hence total space occupied by grid can be estimated as:

$$\begin{aligned} S_{grid} &= S_{cells} + S_{lists} \\ &= \frac{2S_p[1 + m(q+c)^2]}{c^2} \end{aligned}$$

General Case: Assume each query on average ends up in L lists (*full* or *part*). Then m queries end up in mL lists. Since the total number of cells is $\frac{1}{c^2}$, each cell has the combined list size of mLc^2 on average.

The space occupied by the lists S_{lists} can be computed as the number of cells times the space needed for *full* and *part* lists per cell on average. As before, $S_{lists} = N_{lists} \times 2S_p = mL \times 2S_p$. Notice S_{cells} remains the same as above. Hence total space occupied by grid in general case can be estimated as:

$$\begin{aligned} S_{grid} &= S_{cells} + S_{lists} \\ &= 2S_p(\frac{1}{c^2} + mL) \end{aligned}$$

For example, for the Pentium III machine, for grid of size 1000 (e.g., $c = 0.001$), for $m = 25,000$ queries and given that $S_p = 4$, the size of grid is: $S_{grid} = 2S_p(\frac{1}{c^2} + mL) = 8(1,000,000 + 25,000 \times L) = 8,000,000 + 200,000 \times L$ bytes. If each query ends up in $L = 1000$ lists on average (i.e., cover 1000 cells – it is normally much smaller, around 100) then $S_{grid} = 208,000,000$ bytes (well within the memory of a typical PIII machine, e.g. the PIII machine we used for our testing has 2 GB).

3.2.6 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 object's current location.

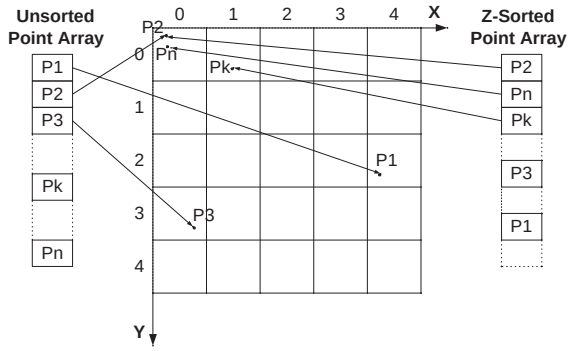


Figure 3.5 Example of un-sorted and z-sorted object arrays

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 3.5. 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 3.5. 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 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.

For example for skewed data, 1,000,000 moving objects and 25,000 of continuous range queries (size 0.01×0.01), the CPU cost of Z-sorting is 4.585 seconds. If the

data is not Z-sorted the processing cost is 1.715 seconds, for Z-sorted data it is 0.763, i.e., in this case there is 2.25× improvement. The cost of Z-sorting corresponds to the processing cost of 2.67 cycles of unsorted data and 6 cycles of sorted data. In other words, by losing a handful of processing cycles for doing the sorting, the performance can be improved 2.25 times.

Of course, the performance will deteriorate as object move and the data will need to be resorted periodically to maintain the level of performance close to that of ideally sorted data.

3.3 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.

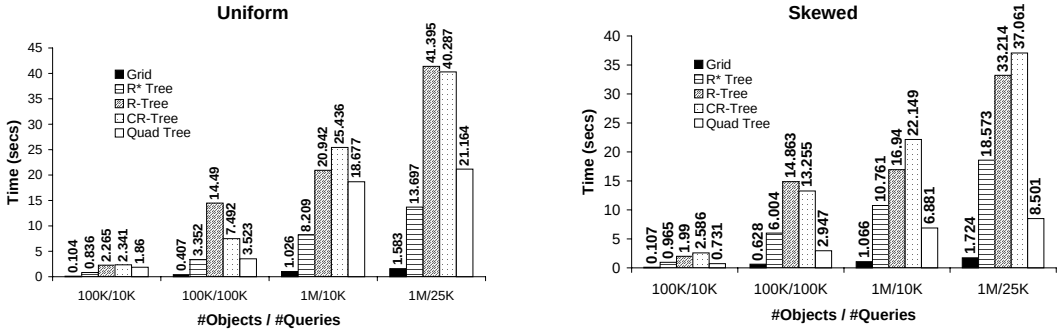
2. **Skewed**: The objects and queries are distributed among five clusters. Within each cluster objects and queries are distributed normally with a standard deviation of 0.05 for objects and 0.1 for queries.

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.

3.3.1 Comparing efficiency of indexes

Figure 3.6a 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 3.6b shows similar results for the skewed case.



(a) (b)

Figure 3.6 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 3.6b 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.

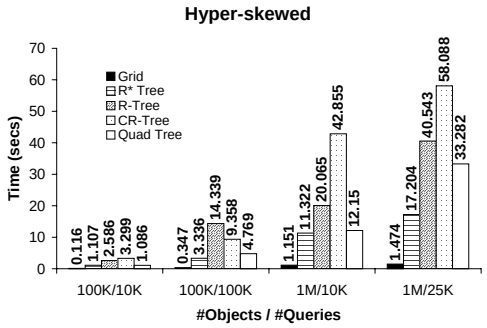


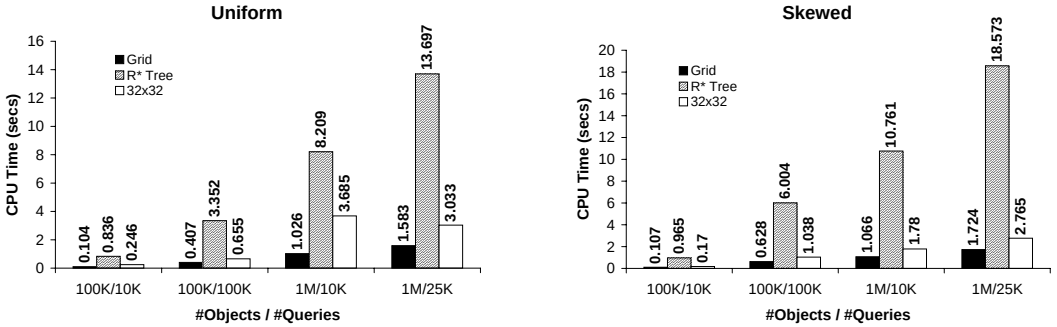
Figure 3.7 Index comparison for Hyper-skewed distribution

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 3.7 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.

3.3.2 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 3.8 Performance of 32-tree (a) Uniform distribution (b) Skewed distribution

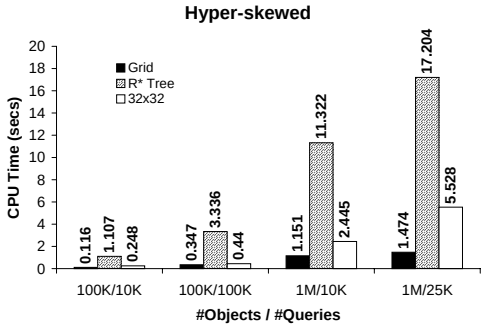


Figure 3.9 Performance of 32-tree for Hyper-skewed distribution

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 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 3.11 and 3.12. 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.

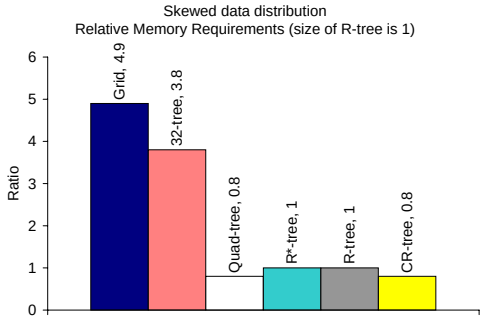
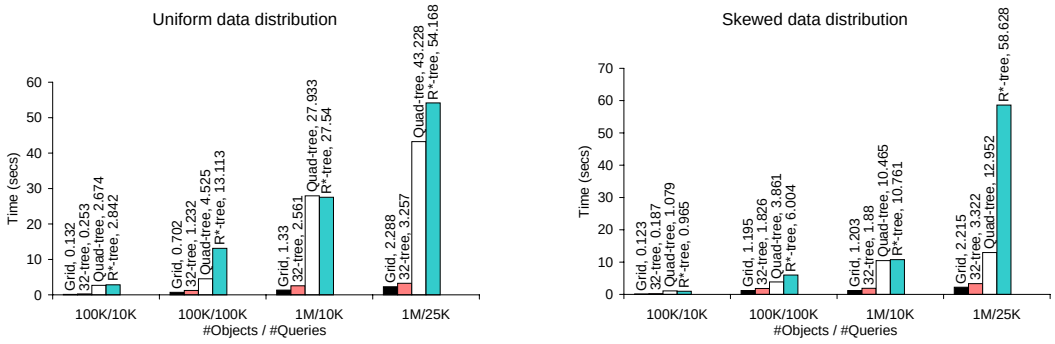


Figure 3.10 Memory requirements relative to the R-tree

Figure 3.10 show relative to the R-tree memory requirements for various indexes. It shows that there is a tradeoff between the amount of memory occupied by an index and its performance, i.e., the fastest indexes require more memory. Even though the Grid index requires the most amount of memory, it was shown in Section 3.2.5 that the grid fits in memory of an average workstation.

3.3.3 Prolonged queries

In this section we present the results of testing with *prolonged* queries of the four best schemes: the Grid, 32-tree, Quad-tree, and R*-tree. In all experiments in this section half of the queries are of the size 0.1×0.001 and the other half is of the size 0.001×0.1. That is the query shape is skewed: namely one side is 100 times larger than the other. As before, our domain represents $1,000 \times 1,000$ mile area, hence the size of each query is 1 mile by 100 miles. We think such a skew in query size is unrealistic and uncommon in real life. Nevertheless this section shows the robustness of our techniques even for prolonged queries.



(a) (b)

Figure 3.11 Performance for 50/50 mix of 0.1×0.001 and 0.001×0.1 queries (a) Uniform (b) Skewed

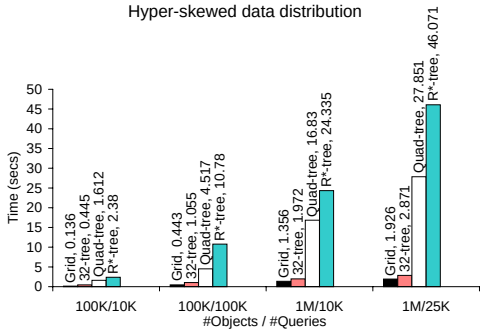


Figure 3.12 Performance for 50/50 mix of 0.1×0.001 and 0.001×0.1 queries: Hyper-skewed distribution

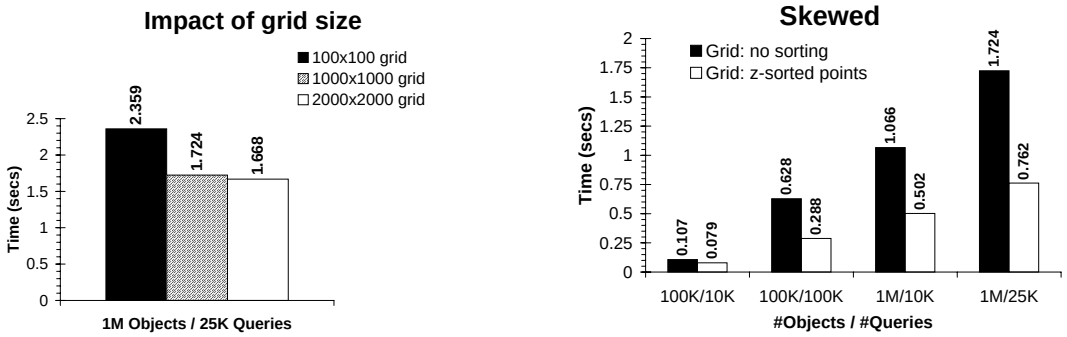
All the figures in this case show similar patters to that of 0.01×0.01 square queries. Notice that the area of each query in this section is equal to the area of a square query from the other experiments. Once again the Grid index substantially outperforms all other indexes. Because a prolonged query is likely to end up in larger amount of cell lists the performance of the Grid index is slightly worse than that of the case with the square queries. The performance of the R*-tree deteriorates substantially because the MBRs are likely to become larger, i.e. each MBR is at least of size 0.1 in one dimension. This leads to larger overlap among the MBRs and, consequently, poorer performance. The performance of the space partitioning quad-tree and 32-tree

is affected adversely too because each query is likely to overlap more partitions, and correspondingly be present in a larger number of nodes.

This section establishes the fact that the superiority of the Grid index verses the other indexes is even greater when the queries are not squares but rather prolonged rectangles with one side substantially greater than the other.

3.3.4 Choice of grid size

In this experiment we study the impact of the number of cells in the grid. The analysis in Section 3.2.4 predicted that a 1000×1000 grid should be chosen for the settings of this example (uniform, 1M objects, 25K queries). Figure 3.13a presents



(a) (b)

Figure 3.13 a) Impact of grid size on processing time. b) Effectiveness of Z-Sorting.

the processing time needed with grid sizes 100×100, 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.

3.3.5 Z-sort optimization

Figure 3.13b 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 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.

3.3.6 Grid: performance of adding and removing queries

We now study the efficiency of modifying the grid index. The results in Figure 3.14 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 is done very efficiently with the grid. For

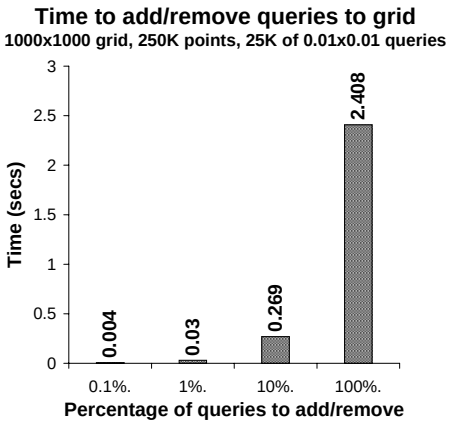


Figure 3.14 Adding and deleting queries.

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.

3.4 Summary

In this chapter 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.

4. GRID-BASED SIMILARITY JOINS

4.1 Introduction

Similarity (spatial) joins are an important database operation for several applications including GIS, multimedia databases, data mining, location-based applications, and time-series analysis. Spatial joins are natural for geographic information systems and moving object environments where pairs of objects located close to each other are to be identified [PD96, LR96]. Many algorithms for basic data mining operations such as clustering [GRS98], outlier detection [KN98], and association rule mining [KH95] require the processing of all pairs of points within a certain distance to each other [BBKK01]. Thus a similarity join can serve as the first step for many of these operations [BBBK00].

The problem of efficient computation of similarity joins has been addressed by several researchers. Most researchers have focused their attention on disk-based joins for high-dimensional data. Current high-end workstations have enough memory to handle joins even for large amounts of data. For example, the self-join of 1 million 32-dimensional data points, using an algorithm similar to that of [BBKK01] (assuming $float$ data type for coordinate and int for point identities) requires roughly 132MB of memory (i.e., $(32 \times 4 + 4) \times 10^6 \approx 132$MB, plus memory for stack etc.). Furthermore there are situations when it is necessary to join intermediate results situated in main memory or sensor data, that is to be kept in main memory. With the availability of a large main memory, disk-based algorithms may not necessarily be the best choice. Moreover, for certain applications (e.g., moving object environments) near real-time computation may be critical and require main memory evaluation.

In this chapter we consider the problem of main memory processing of similarity joins, also known as ε-joins. Given two datasets A and B of d-dimensional points and

value $\varepsilon \in \Re$, the goal of a join operation is to identify all pairs of points, one from each set, that are within distance ε from each other, i.e.

$$A \bowtie_{\varepsilon} B = \{(a, b) : \|a - b\| < \varepsilon;\ a \in A,\ b \in B\}.$$

While several research efforts have concentrated on designing efficient high-dimensional join algorithms, the question of which method should be used when joining low-dimensional (e.g., 2–6 dimensions) data remains open. This chapter addresses this question and investigates the choice of join algorithm for low- and high-dimensional data. We introduce two new join algorithms: the *Grid-join* and *EGO*-join*, and evaluate their along with the state of the art algorithm – EGO-join [BBKK01], and a method which serves as a benchmark in many similar publications, the RSJ join [BKS93].

These techniques are compared through experiments using synthetic and real data. We considered the total wall-clock time for performing a join without ignoring any costs, such as pre-sorting data, building/maintaining index etc. The experimental results show that the Grid-join approach showed the best results for low-dimensional data.

Under the Grid-join approach, the join of two sets A and B is computed using an index nested loop approach: an index (i.e., specifically constructed 2-dimensional grid) is built on circles with radius ε centered at the first two coordinates of points from set B. The first two coordinates of points from set A are used as point-queries to the grid-index in order to compute the join. Although several choices are available for constructing this index, only the grid is considered in this chapter. The choice is not accidental, it is based upon our earlier results for main memory evaluation of range queries. In Chapter 3 we have shown that for range queries over moving objects, using a grid index results in an order of magnitude better performance than memory optimized R-tree, CR-tree, R*-tree, or Quad-tree.

The results for high-dimensional data show that the EGO*-join is the best choice of join method, unless ε is very small. The EGO*-join that we propose in this chapter is based upon the EGO-join algorithm. The Epsilon Grid Order (EGO) join

[BBKK01] algorithm was shown to outperform other techniques for spatial joins of high-dimensional data. The new algorithm significantly outperforms EGO-join for all cases considered. The improvement is especially noticeable when the number of dimensions is not very high, or the value of ε is not large. The RSJ algorithm is significantly poorer than all other three algorithms in all experiments. In order to join two sets using RSJ, an R-tree index needs to be built or maintained on both of these sets. But unlike the case of some approaches, these indexes need not be rebuilt when the join is recomputed with different ε.

Although not often addressed in related research, the choice of the ε parameter for the join is critical to producing meaningful results. We have discovered that often in similar research the choice of values of ε yields very small selectivity, i.e. almost no point from one dataset joins with a point from the other dataset. In Section 4.4.1 we present a discussion on how to choose appropriate values of ε.

The contributions of this chapter are as follows:

- Two join algorithms that give better performance (almost an order of magnitude better for low dimensions) than the state of the art EGO-join algorithm.
- Recommendations for the choice of join algorithm based upon data dimensionality d, and ε.
- Highlight the importance of the choice of ε and the corresponding selectivity for experimental evaluation.
- Highlight the importance of the cache miss reduction techniques: spatial sortings (2.5 times speedup) and clustering via utilization of dynamic arrays (40% improvement).
- For the Grid-join, the choice of grid size is an important parameter. In order to choose good values for this parameter, we develop highly accurate estimator functions for the cost of the Grid-join. These functions are used to choose an optimal grid size.

The rest of this chapter is organized as follows. Related work is discussed in Section 4.5. The new Grid-join and EGO*-join algorithms are presented in Section 4.2. The proposed join algorithms are evaluated in Section 4.4, and Section 4.6 concludes the chapter. A sketch of the algorithm for selecting grid size and cost estimator functions for Grid-join are presented in Section 4.3.

4.2 Similarity join algorithms

In this section we introduce EGO*-join and Grid-join. The EGO*-join method is discussed in Section 4.2.1. In Section 4.2.2 we first present the Grid-join algorithm followed by an important optimization for improving the cache hit-rate. An analysis of the appropriate grid size as well as cost prediction functions for the Grid-join is presented in Section 4.3.

4.2.1 EGO*-join (J_{EGO*})

In this section we present an improvement of the disk-based EGO-join algorithm proposed in [BBKK01]. We dub the new algorithm the EGO*-join. We use notation J_{EGO} for the EGO-join procedure and J_{EGO*} for the EGO*-join procedure. According to [BBKK01], the state of the art algorithm J_{EGO} was shown to outperform other methods for joining massive, high-dimensional data.

We begin by briefly describing J_{EGO} as presented in [BBKK01] followed by our improvement of J_{EGO}.

The Epsilon Grid Order: J_{EGO} is based on the so called Epsilon Grid Ordering (EGO), see [BBKK01] for details. In order to impose an EGO on dataset A, a regular grid with the cell size of ε is laid over the data space. The grid is imaginary, and never materialized. For each point in A, its cell-coordinate can be determined in O(1) time. A lexicographical order is imposed on each cell by choosing an order for the dimensions. The EGO of two points is determined by the lexicographical order of the corresponding cells that the points belong to.

Input: Datasets A, B, and $\varepsilon \in \Re$

Output: Result set R

1. EGO-sort(A, ε)
2. EGO-sort(B, ε)
3. join_sequences(A, B)

Figure 4.1 EGO-join Procedure, J_{EGO}

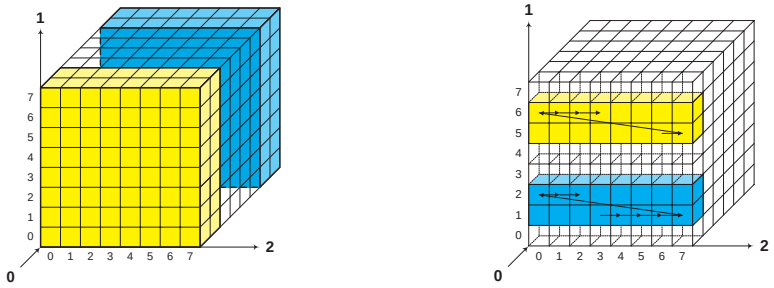
EGO-sort: In order to perform J_{EGO} of two sets A and B with a certain ε, first the points in these sets are sorted in accordance with the EGO for the given ε. Notice that for a subsequent J_{EGO} operation with a different ε sets A and B need to be sorted again since their EGO values depend upon the cells.

Recursive join: The procedure for joining two sequences is recursive. Each sequence is further subdivided into two roughly equal subsequences and each subsequence is joined recursively with both its counterparts. The partitioning is carried out until the length of both subsequences is smaller than a threshold value, at which point a simple-join is performed. In order to avoid excessive computation, the algorithm avoids joining sequences that are guaranteed not to have any points within distance ε of each other. Such sequences can be termed *non-joinable*.

EGO-heuristic: A key element of J_{EGO} is the heuristic used to identify *non-joinable* sequences. The heuristic is based on the number of inactive dimensions, which will be explained shortly. To understand the heuristic, let us consider a simple example. For a short sequence its first and last points are likely to have the same first cell-coordinates. For example, points with corresponding cell-coordinates $(2, 7, 4, 1)$ and $(2, 7, 6, 1)$ have two common prefix coordinates $(2, 7, \times, \times)$. Their third coordinates differ – this corresponds to the *active* dimension, the first two dimensions are

called *inactive*. This in turn means that for this sequence all points have 2 and 7 as their first two cell-coordinates – because both sequences are EGO-sorted before being joined.

The heuristic first determines the number of inactive dimensions for both sequences, and computes *min* – the minimum of the two numbers. It is easy to prove that if there is a dimension between 0 and $min - 1$ such that the cell-coordinates of the first points of the two sequences differ by at least two in that dimension, then the sequences are non-joinable. This is based upon the fact that the length of each cell is ε.



(a) (b)

Figure 4.2 Two sequences with (a) 0 inactive dimensions (b) 1 inactive dimension. Unlike EGO-heuristic, in both cases EGO*-heuristic is able to tell that the sequences are non-joinable.

New EGO*-heuristic: The proposed J_{EGO*} (EGO*-join) algorithm is J_{EGO} (EGO-join) with an important change to the heuristic for determining that two sequences are non-joinable. The use of the EGO*-heuristic significantly improves performance of the join, as will be seen in Section 4.4.

We now present our heuristic with the help of an example for which J_{EGO} is unable to detect that the sequences are *non-joinable*.

Two sequences are shown in Figure 4.2(b). Assume that each sequence has many points. One sequence starts in cell (0,1,3) and ends in cell (0,2,2). The second sequence starts in cell (0,5,6) and ends in (0,6,3). Both sequences have one inactive

---

Input: The first and last cells of a sequence: C_F andC_L

Output: Bounding rectangle BR

1. **for** $i \leftarrow 0$ **to** $d-1$ **do**
   (a) $BR.lo[i] \leftarrow C_F.x[i]$
   (b) $BR.hi[i] \leftarrow C_L.x[i]$
   (c) **if** $(R.lo[i] = R.hi[i])$ **then continue**
   (d) **for** $j \leftarrow i+1$ **to** $d-1$ **do**
      i. $BR.lo[j] \leftarrow 0$
      i. $BR.hi[j] \leftarrow MAX\_CELL$
   (e) **break**
2. **return** BR

Figure 4.3 J_{EGO*}: procedure for obtaining a Bounding Rectangle of a sequence

dimension: 0. The EGO-heuristic will conclude that these two should be joined, allowing recursion to proceed. Figure 4.2(a) demonstrates the case when two sequences are located in two separate slabs, both of which have the size of at least two in each dimension. There are no inactive dimensions for this case and recursion will proceed further for J_{EGO}.

The new heuristic being proposed is able to correctly determine that for the cases depicted in Figures 4.2(a) and 4.2(b) the two sequences are *non-joinable*. It should become clear later on that, in essence, our heuristic utilizes not only inactive dimensions but also the active dimension.

The heuristic uses the notion of a Bounding Rectangle for each sequence. Notice that in general, given only the first and last cells of a sequence, it is impossible to compute the Minimum Bounding Rectangle (MBR) for the sequence. However, it is

possible to compute a Bounding Rectangle (BR). Figure 4.3 describes an algorithm for computing a bounding rectangle.

The procedure takes as input the coordinates for first and last cells of the sequence and produces the bounding rectangle as output. To understand the getBR() algorithm, note that if first and the last cell have n prefix equal coordinates (e.g., $(1, 2, 3, 4)$ and $(1, 2, 9, 4)$ have two equal first coordinates – $(1, 2, \times, \times)$) then all cells of the sequences have the same values in the first n coordinates (e.g., $(1, 2, \times, \times)$ for our example). This means that the first n coordinates of the sequence can be bounded by that value. Furthermore, the active dimension can be bounded by the coordinates of first and last cell in that dimension respectively. Continuing with our example, the lower bound is now $(1, 2, 3, \times)$ and the upper bound is $(1, 2, 9, \times)$. In general, we cannot say anything precise about the rest of the dimensions, however the lower bound can always be set to 0 and upper bound to MAX_CELL.

Input: Two sequences A and B

Output: Result set R

1. $BR_1 \leftarrow \text{getBR}(A.first,\ A.last)$

1. $BR_2 \leftarrow \text{getBR}(B.first,\ B.last)$

3. Expand BR_1 by one in all directions

4. **if** $(BR_1 \cap BR_2 = \emptyset)$ **then return** $\emptyset$

5. ... // continue as in J_{EGO}

Figure 4.4 Beginning of J_{EGO*}: EGO*-heuristic

Once the bounding rectangles for both sequences being joined are known, it is easy to see that if one BR, expanded by one in all directions, does not intersect with the other BR, than the two sequences will not join.

As we shall see in Section 4.4, J_{EGO*} significantly outperform J_{EGO} in all instances. This improvement is a direct result of the large reduction of the number of sequences compared based upon the above criterion. This result is predictable since if EGO-heuristic can recognize two sequences as non-joinable then EGO*-heuristic will always do the same, but if the EGO*-heuristic can recognize two sequences as non-joinable then, in general, there are many cases when the EGO-heuristic will decide the sequences are joinable. Thus the EGO*-heuristic is more powerful. Furthermore, the difference in CPU time needed to compute the heuristics given the same two sequences is insignificant.

4.2.2 Grid-join (J_G)

Assume for now that we are dealing with 2-dimensional data. The spatial join of two datasets, A and B, can be computed using a standard Index Nested Loop approach as follows. We treat one of the point data sets as a collection of circles of radius ε centered at each point of one of the two sets (say B). This collection of circles is then indexed using some spatial index structure. The join is computed by taking each point from the other data set (A) and querying the index on the circles to find those circles that contain the query point. Each point (from B) corresponding to each such circle joins with the query point (from A). An advantage of this approach (as opposed to the alternative of building an index on the points of one set and processing a circle region query for each point from the other set) is that point queries are much simpler than region queries and thus tend to be faster. For example, a region query on a quad-tree index might need to evaluate several paths while a point query is guaranteed to be a single path query. An important question is the choice of index structure for the circles.

In Chapter 3 we have investigated the execution of large numbers of range queries over point data in the context of evaluating multiple concurrent continuous range queries on moving objects. The approach can also be used for spatial join if we compute the join using the Index Nested Loops technique mentioned above. The two approaches differ only in the shape of the queries which are circles for the spatial join problem and rectangles for the range queries.

In Chapter 3 the choice of a good main-memory index was investigated. Several key index structures including R-tree, R*-tree, CR-tree [KCK01], quad-tree, and 32-tree were considered. All trees were optimized for main memory. The conclusion of the study was that a simple one-level Grid-index outperformed all other indexes by almost an order of magnitude for uniform as well as skewed data. Due to its superior performance, in this study, we use the Grid-index for indexing the ε-circles.

The Grid Index While many variations exist, we have designed our own implementation of the Grid-index, which we denote as I_G. I_G is built on circles with ε-radius. Note however, that it is not necessary to generate a new dataset consisting of these circles. Since each circle has the same radius (ε), the dataset of the points representing the centers of these circles is sufficient. The similarity join algorithm which utilizes I_G is called the Grid-join, or J_G for short.

Case of 2 dimensions For ease of explanation assume the case of 2-dimensional data. I_G is a 2-dimensional array of cells. Each cell represents a region of space

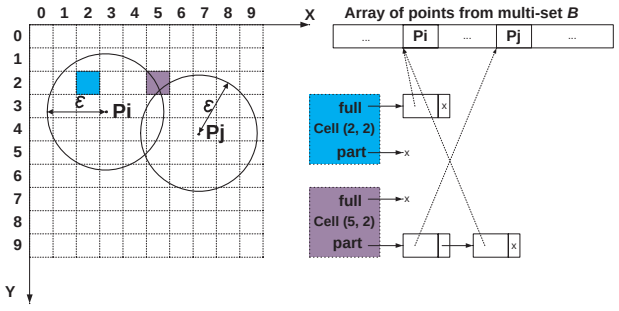


Figure 4.5 An example of the Grid Index, I_G

generated by partitioning the domain using a regular grid.

Figure 4.5 shows an example of I_G. Throughout the chapter, we assume that the domain is normalized to the unit d-dimensional hyper-cube, $[0,1]^d$. In this example, the domain is divided into a 10×10 grid of 100 cells, each of size 0.1×0.1.

Since the grid is uniform, it is easy to calculate cell-coordinates of an object in O(1) time. Each cell contains two lists that are identified as *full* and *part*, as shown in Figure 4.5. Let $C(p,r)$ denote a circle with center at point p and radius r The *full* (*part*) list of a cell contains *pointers* to all points b_i from B such that $C(b_i,\varepsilon)$ fully (partially) cover the cell. That is for cell C in I_G its *part* and *full* lists can be represented mathematically as $C.full = \{b : C \subset C(b,\varepsilon);\ b \in B\}$ and $C.part = \{b : C \not\subset C(b,\varepsilon) \wedge C \cap C(b,\varepsilon) \neq \emptyset;\ b \in B\}$.

To find all points within ε-distance from a given point a first the cell corresponding to a is retrieved. All points in *full* list are guaranteed to be within ε-distance. Points in *part* list need to be post-processed.

The choice of data structures for the *full* and *part* lists is critical for performance. We implemented these lists as dynamic-arrays[1] rather than lists which improves performance by roughly 40% due to the resulting clustering (and thereby reduced cache misses).

Case of d dimensions For the general d-dimensional case, the first 2 coordinates of points are used for all operations exactly as in 2-dimensional case except for the processing of *part* lists, which uses all d coordinates to determine whether $\|a-b\| < \varepsilon$.

The reason for two separate lists per cell for 2-dimensional points is that points in the *full* list do not need potentially costly checks for relevance since they are guaranteed to be within ε-distance. Keeping a separate *full* list is of little value for more than 2 dimensions since now it too needs post-processing to eliminate false positives similar to the *part* list. Therefore only one list is kept for all circles that at least partially intersect the cell in the chosen 2 dimensions. We call this list *part* list: $C.part = \{b : C \cap C(b',\varepsilon) \neq \emptyset;\ b \in B\}$.

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

Input: Datasets A, B, and $\varepsilon \in \Re$

Output: Result set R

1. $R \leftarrow \emptyset$
2. z-sort(A)
3. z-sort(B)
4. Initialize I_G
5. **for** $i \leftarrow 0$ **to** $|B| - 1$ **do**
   (a) $b'_i \leftarrow (b_i^0, b_i^1)$
   (b) Insert $\{b_i, C(b'_i, \varepsilon)\}$ into I_G
6. **for** $i \leftarrow 0$ **to** $|A| - 1$ **do**
   (a) $a'_i \leftarrow (a_i^0, a_i^1)$
   (b) Let C_i be the cell in I_G corresponding to a'_i
   (c) **for** $j \leftarrow 0$ **to** $|C.part| - 1$ **do**
      i. $b \leftarrow C_i.part[j]$
      ii. **if** $(\|a_i - b\| < \varepsilon)$ **then** $R \leftarrow R \cup \{(a_i, b)\}$
   (d′) **for** $j \leftarrow 0$ **to** $|C_i.full| - 1$ **do**
      i. $b \leftarrow C_i.full[j]$
      ii. $R \leftarrow R \cup \{(a_i, b)\}$
7. **return** R

Figure 4.6 Grid-join procedure, J_G

J_G is described in Figure 4.6. Steps 2 and 3, the z-sort steps, apply a spatial sort to the two datasets. The need for this step is explained later. I_G is initialized in Step 4. In the loop in Step 5, all points b_i from set B are added to I_G one by one. First b'_i, a 2-dimensional point constructed from the first two coordinates of b_i, is

considered. Then pointer to b_i is added to *part* lists of each cell C in I_G that satisfies $C \cap C(b'_i, \varepsilon) \neq \emptyset$.

The loop in Step 6 performs a nested loop join. For each point a_i in A all points from B that are within ε distance are determined using I_G. To do this, point a'_i is constructed from the first two coordinates of a_i and the cell corresponding to a'_i in I_G, C_i, is determined in Steps 6(a) and 6(b). Then, in Step 6(c), the algorithm iterates though all elements of the *part* list of cell C_i and finds all relevant to a points. Step 6(d′) is analogous to Step 6(c) and valid only for 2-dimensional case.

Choice of grid size The performance of J_G depends on the choice of grid size, therefore it must be selected carefully. Intuitively, the finer the grid the faster the processing but the slower the time needed to initialize the index and load the data into it. We now present a sketch of a solution for selecting appropriate grid size.

The first step is to develop a set of estimator functions that predict the cost of the join given a grid size. The cost is composed of three components, the costs of: (a) initializing the empty grid; (b) loading the dataset B into the index; and (c) processing each point of dataset A through this index. Section 4.3 presents details on how each of these costs is estimated. The quality of the prediction of these functions was found to be extremely high. Using these functions, it is possible to determine which grid size would be optimal. These functions can also be used by a query optimizer – for example to evaluate whether it would be efficient to use either J_G for the given parameters or another method of joining data.

Improving the cache hit-rate The performance of main-memory algorithms is greatly affected by cache hit rates. In this section we describe an optimization that improves cache hit rates and, consequently, the overall performance of J_G. This optimization is similar to that of Section 3.2.6.

Sorting the points to ensure that points that are close to each other are also close in the array order can easily be achieved by various methods. As in Section 3.2.6 we choose to use a sorting based on the Z-order. The speed-up is achieved because such points are more likely to be covered by the same circles than points that are

far apart, thus the relevant information is more likely to be retrieved from the cache rather than from main memory. We sort not only set A but also set B, which reduces the time needed to add circles to I_G. As we will see in the Experimental Section, $\sim$2.5$\times$ speedup is achieved by utilizing Z-sort, e.g. as shown in Figure 4.18.

4.3 Choice of grid size

In this section we develop cost estimator functions for J_G. These functions can be used to determine the appropriate choice of grid size for computing the similarity join for a specific problem. The discussion focuses on the case of two dimensions, but can be generalized to any number of dimensions in a straight-forward manner.

Table 4.1 lists parameters needed for our analysis. All the parameters are known before the join, except for the grid size n. We are interested in finding n such that

Table 4.1 Parameters for J_G cost estimation

Parameter	*Meaning*
A	First dataset for join
B	Second dataset, (on which the index is built)
$k = \|A\|$	Cardinality of dataset A
$m = \|B\|$	Cardinality of dataset B
c	Length of side of a cell
$n = \frac{1}{c}$	Grid size: i.e. $n \times n$ grid
eps, ε	Epsilon parameter for the join

the time needed for the join is minimized. Furthermore, if there are several values of n that yield minimal or close to minimal join cost, then we are interested in the smallest such n. This is because the memory requirements for the grid increase with the number of cells in the grid.

In order to determine the relationship between the join cost and the various parameters of the problem, we develop what we call estimator (or predictor) functions for the various phases of J_G. Once the predictor functions are constructed, a suitable choice for n can be found by identifying a minimal value of the cost. For the value of n selected, the predictor functions are also useful in providing an estimated cost to the query optimizer which can use this information to decide whether or not J_G should be used for the problem.

In our analysis we assume the uniform distribution of points in set A and B. The J_G procedure can be divided into three phases:

1. The **initialization** phase: initialization of the grid pointers and lists
2. The **loading** phase: loading the data into the grid
3. The **processing** phase: processing the point queries using the grid

4.3.1 The trade-off between costs of the *building* and *processing* phases

The *initialization* and *loading* phases collectively are called the *building* index phase. There is a tradeoff between the *building* and *processing* phases with respect to the grid size n. With smaller grid size n (and thus fewer cells), each circle is likely to intersect with fewer cells and thus be added to fewer full and part lists, making the building phase faster. On the other hand, with fewer cells the average length of the part lists is likely to be larger and each query takes longer to process on average. In other words, the coarser (i.e., smaller n) the grid the faster the *building* phase, but the slower the *processing* phase.

4.3.2 Determining domain of grid size n

While the general trend is that a finer grid would imply shorter query processing time (since the part lists would be shorter or empty), beyond a certain point, a finer grid may not noticeably improve performance. For our implementation, 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. It is enough to choose the grid size n such that the average size of part list is one; further partitioning does not noticeably improve query processing time. Thus we can estimate an upper bound U of the domain $\mathcal{D}_n = [1, U]$ of the good choice for grid size n.

For example, for 2-dimensional square queries it can be shown that the upper bound is given by (see Chapter 3):

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

In this formula q is the size of each square. Since for similarity join we are adding circles, the formulas is reused by approximating the circle by a square with the same area (i.e., $q \approx \varepsilon\sqrt{\pi}$). The corresponding formula for U is therefore:

$$U = \begin{cases} \lceil 4\sqrt{\pi}\varepsilon m \rceil & \text{if } \varepsilon > \frac{1}{2\sqrt{\pi m}}; \\ \lceil \frac{1}{\frac{1}{\sqrt{m}} - \varepsilon\sqrt{\pi}} \rceil & \text{otherwise.} \end{cases}$$

A finer grid than that specified by the above formula gives very minor performance improvement while incurring a large memory penalty. Thus the formula establishes the upper bound for grid size domain. However, if the value returned by the formula is too large, the grid might not fit in memory. In that case the upper bound U is further limited by memory space availability.

In our experiments the optimal value for the grid size tended to be closer to one rather than to U (remember, $\mathcal{D}_n = [1, U]$), as shown in Figures 4.10 and 4.11.

4.3.3 Method of analysis

For each of the phases of J_G, the analysis is conducted as follows. 1) First the parameters on which a phase depends are determined. 2) The nature of the dependence on each parameter separately is predicted based on the algorithm and implementation of J_G. Since the grid is a simple data structure, dependence on a parameter, as a rule, is not complicated. 3) Next the dependence on the combination of the parameters is predicted based on the dependence for each parameter. 4) Finally, an explanation

is given on how the calibration of predictor functions can be achieved on a specific machine.

4.3.4 Estimating cost of the *initialization* phase

The time to initialize the index depends only on the grid size n. The process of index initialization can be described as $O(1)$ operations followed by the initialization of n^2 cells. Thus the index initialization time is expected to be a polynomial of degree two over n: $P_{init}(n) = an^2 + bn + c$, for some coefficients a, b, and c. This value of the coefficients depend upon the particular machine on which the initialization is performed. They can be determined through a calibration step. To validate the correctness of this estimator, we calibrated it for a given machine. The corresponding estimator function was then used to predict the performance for other values of n not used for the calibration. The result is shown in Figures 4.7 and 4.8, where

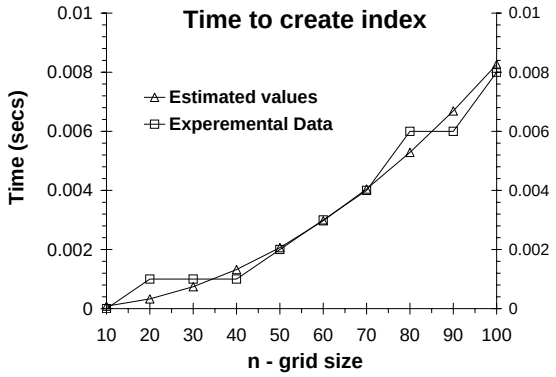


Figure 4.7 Estimating cost of the *initialization* phase, $n \in [10, 100]$

$a = 8.26 \times 10^{-7}$, $b = 0$, and $c = 0$. The two figures shown are for different ranges of the grid size n: on the left n varies from 10 to 100, on the right n varies from 100 to 1000. The figures show the actual times measured for different values of n as well as the time predicted by the estimator function. As can be seen, the estimator gives a

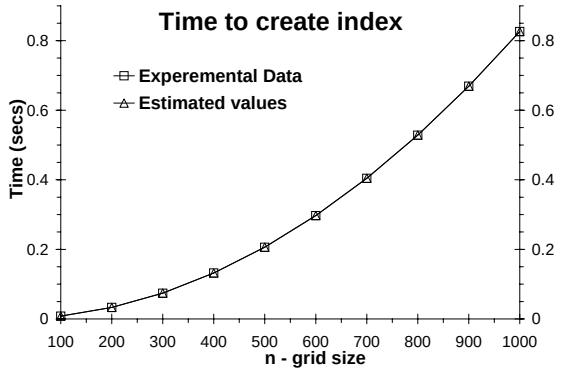


Figure 4.8 Estimating cost of the *initialization* phase, $n \in [100, 1000]$

very good approximation of the actual initialization times. This is especially true for larger values of n.

Figures 4.7 and 4.8 show that the time needed for index initialization phase can be approximated well with a simple polynomial. Any numerical method can be used for calibrating the coefficients a, b, and c for a particular machine.

4.3.5 Estimating cost of the *loading* phase

This phase is more complicated than the initialization phase because its cost depends on three parameters: the grid size n, cardinality of the indexed set B (i.e., m), and ε. By analyzing the dependence on each parameter separately, we have determined that the overall function can be represented as a polynomial $P_{add}(n, m, \varepsilon) = a_{17}n^2\varepsilon^2 m + \cdots + a_1 m + a_0$ with degrees of n and ε no greater than two and degree of m no greater than one. The next step is to calibrate the coefficients a_i's. This can be done by solving a system of 18 linear equations. These equations can be obtained by choosing three different values of n, three values of ε, and two values of m (i.e., $3 \times 3 \times 2 = 18$).

In our example the combinations of the following calibration points have been examined in order to get the coefficients: three values for the grid size n : $n_0 = 10$, $n_1 = 100$, $n_2 = 200$; three values for ε : $\varepsilon_0 = 0.001$, $\varepsilon_1 = 0.01$, $\varepsilon_2 = 0.02$; and two

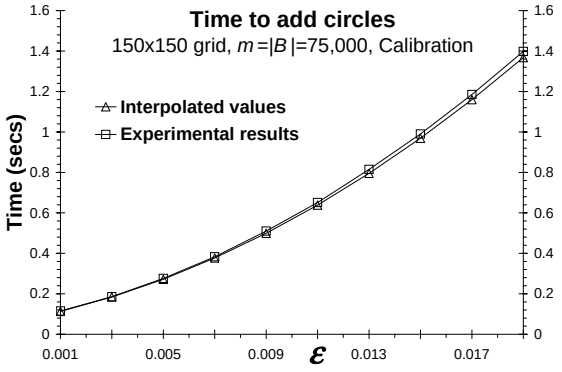


Figure 4.9 Estimating cost of the *loading* phase

values for B's cardinality m : $m_0 = 50\,000$, and $m_1 = 100\,000$. The choice of values implies we assume that typically for the grid size $n \in [10, 200]$, $\varepsilon \in [0.001, 0.02]$, and B's cardinality $m \in [50\,000, 100\,000]$. The linear system was solved using the "Gaussian elimination with pivoting" method. Figure 4.9 demonstrates time needed for the loading phase as function of ε when $n = 150$ and $m = 75\,000$; the other curve is our interpolation polynomial. Observe that the estimator function is highly accurate. In fact we have never encountered more than a 3% relative error in our experiments.

4.3.6 Estimating cost of the *processing* phase

The processing phase depends on all parameters: the grid size n, $k = |A|$, $m = |B|$, and ε. Thankfully, the dependence on k is linear since each point is processed independent of other points. Once the solution for some fixed k_0 is known, it is easy to compute the solution for an arbitrary k. However, there is a small complication: the average lengths of the *full* and *part* lists are given by different formulae depending upon whether cell size c is greater than $\sqrt{\pi}\varepsilon$ or not (see Chapter 3, in our case query side size q is replaced by $\sqrt{\pi}\varepsilon$).

Consequently the *processing* phase cost, when A's cardinality k is fixed at some value k_0, can be estimated by one of two polynomials, depending upon whether $\sqrt{\pi}\varepsilon \geq$

c is true: $P_{proc,\sqrt{\pi}\varepsilon \geq c}(c, \varepsilon, m, k_0)$ and $P_{proc,\sqrt{\pi}\varepsilon < c}(c, \varepsilon, m, k_0)$ each of type $P(c, \varepsilon, m, k_0) \equiv a_{17}c^2\varepsilon^2 m + \cdots + a_1 m + a_0$ with degrees of c and ε no greater than two and degree of m no greater than one. Once again the calibration can be done by solving a system of 18 linear equations for each of the two cases.

4.3.7 Estimating total cost of J_G

The estimated total time needed for J_G is the sum of estimated time needed for each phase. Figures 4.10 and 4.11 demonstrate estimation of time needed for J_G

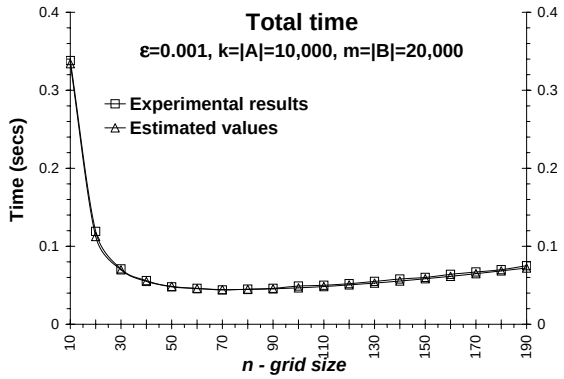


Figure 4.10 Estimation of total join time, $n \in [10, 190]$

when $\varepsilon = 0.001$, $m = 20\,000$, $k = 10\,000$ as a function of grid size n. The estimator functions of each phase were calibrated using different values of m, k, and ε than those shown in the figure.

4.3.7.1 Bisection method

A simple *bisection* method for finding the optimal value of n was used. This method assumes that it is given a concave downwards function, defined on $[a, b]$. The function has been concave downwards in all our experiments, however in future work we hope to prove that the estimator function is always concave downwards for various combinations of parameters. The bisection method in this context works as

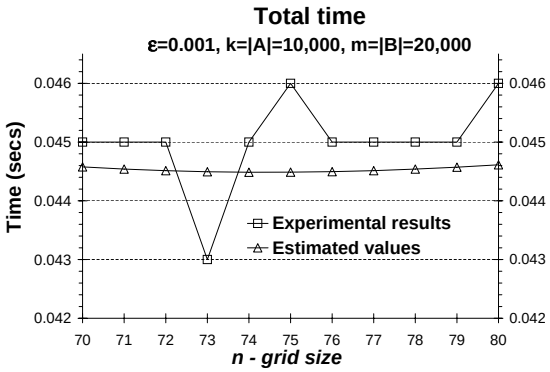


Figure 4.11 Estimation of total join time, $n \in [70, 80]$

follows. The goal is to find the leftmost minimum on the interval $[a, b]$. Compute $c = (a + b)/2$. If $f(c - 1) \leq f(c + 1)$ then make new b be equal c and repeat the process, otherwise make new a be equal c and repeat the process. The process is repeated until $(b - a) < 2$.

The bisection method for the example in Figures 4.10 and 4.11 gives an estimated optimal value for the grid size n as 74. Experimentally, we found that the actual optimal value for n was 73. The difference between time needed for J_G with 73×73 grid and 74×74 grid is just two milliseconds for the given settings. These numbers show the high accuracy of the estimator functions.

4.4 Experimental results

In this section we present the performance results for in-memory joins using J_{RSJ} (RSJ join), J_G, J_{EGO} [BBKK01], and J_{EGO*}. The results report the actual time 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. All multidimensional points were distributed on the unit d-dimensional box $[0,1]^d$. The number of points ranges from 68,000 to 200,000. For distributions of points in the domain we considered the following cases:

1. **Uniform**: Points are uniformly distributed.

2. **Skewed**: The points are distributed among five clusters. Within each cluster points are distributed normally with a standard deviation of 0.05.

3. **Real data**: We tested data from ColorHistogram and ColorMoments files representing image features. The files are available at the UC Irvine repository. ColorMoments stores 9-dimensional data, which we normalized to $[0,1]^9$ domain, ColorHistogram – 32-dimensional data. For experiments with low-dimensional real data, a subset of the leading dimensions from these datasets were used. Unlike uniform and skewed cases, for real data a self-join is done.

Often, in similar research, the costs of sorting the data, building or maintaining the index or costs of other operations needed for a particular implementation of join are ignored. No cost is ignored in our experiments for J_G, J_{EGO}, and J_{EGO*}. One could argue that for J_{RSJ} the two indexes, once built, need not be rebuilt for different ε. While there are many other situations where the two indexes need to be built from scratch for J_{RSJ}, we ignore the cost of building and maintaining indexes for J_{RSJ}, thus giving it an advantage.

4.4.1 Correlation between selectivity and ε

The choice of the parameter ε is critical when performing an ε-join. Little justification for choice of this parameter has been presented in related research. We present this section because we have discovered that often in similar research selected values of ε are too small. We think that the mistake happened because too often researchers choose to test self-join of dataset A which is simpler than join of two different datasets A and B. In self-join each point joins at least with itself. Thus the cardinality of the

result set R is no less than the cardinality of A. By increasing the dimensionality d and fixing ε to a relatively small value, the size needed to store each data point increases, consequently the *size* needed to store R (e.g., in bytes) increases, even though the *cardinality* of R is close to the cardinality of A. The increase of size of R is probably often mistaken for the increase of cardinality of R.

The choice of ε has a significant effect on the selectivity depending upon the dimensionality of the data. The ε-join is a common operation for similarity matching. Typically, for each multidimensional point from set A a few points (i.e., from 0 to 10, possibly from 0 to 100, but unlikely more than 100) from set B need to be identified on average. The average number of points from set B that joins with a point from set A on average is called *selectivity*.

In our experiments, selectivity motivated the range of values chosen for ε. The value of ε is typically lower for smaller number of dimensions and higher for high-dimensional data. For example a 0.1×0.1 square[2] query ($\varepsilon = 0.1$) is 1% of a two-dimensional domain, however it is only 10^{-6}% of an eight-dimensional domain, leading to small selectivity.

Let us estimate what values for ε should be considered for joining high-dimensional uniformly distributed data such that a point from set A joins with a few (close to 1) points from set B. Assume that the cardinality of both sets is m. We need to answer the question: what should the value of ε be such that m hyper-squares of side ε completely fill the unit d-dimensional cube? It is easy to see that the solution is $\varepsilon = \frac{1}{m^{1/d}}$. Figure 4.12 plots this function $\varepsilon(d)$ for two different values of m. Our experimental results for various number of dimensions corroborate the results presented in the figure. For example the figure predicts that to obtain a selectivity close to one for 32-dimensional data, the value of ε should be close to 0.65, or 0.7, and furthermore that values smaller than say 0.3, lead to zero selectivity (or close to zero) which is of little value[3]. This is in very close agreement to the experimental results.

[2] A square query was chosen to demonstrate the idea, ideally one should consider a circle.

[3] For self-join selectivity is always at least 1, thus selectivity 2–100 is desirable.

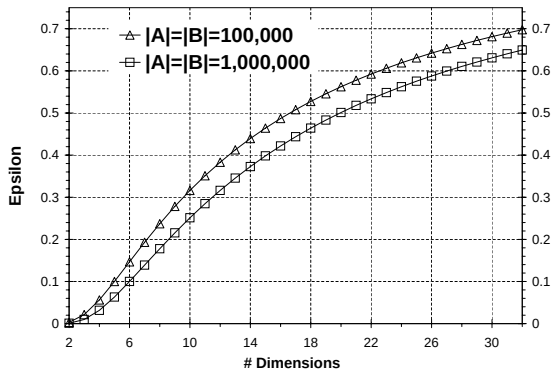


Figure 4.12 Choosing ε for selectivity close to one for 10^5 (and 10^6) points uniformly distributed on $[0,1]^d$

If the domain is not normalized to the unit square, such as in [KS98], the values of ε should be scaled accordingly. For example ε of 0.1 for $[-1,1]^d$ domain correspond to ε of 0.05 for our $[0,1]^d$ domain. Figure 4.13 demonstrates the pitfall of using

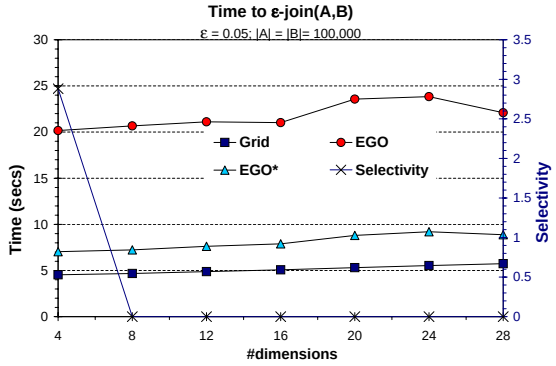


Figure 4.13 Pitfall of using improper selectivity

an improper selectivity. The parameters of the experiment (distribution of data, cardinality of sets and ε (scaled)) are set to the values used in one publication. With this choice of ε the selectivity plunges to zero even for the 10-dimensional case. In fact, for our case, the figure presumably shows that the Grid-join is better than J_{EGO} and

J_{EGO*} even for high-dimensional cases. However, the contrary is true for a meaningful selectivity as will be shown in Section 4.4.3.

Due to the importance of the selectivity in addition to the value of ε, we plot the resulting selectivity in each experiment. The selectivity values are plotted on the y-axis at the right end of each graph. The parameter ε is on the x-axis, and the time taken by each join method is plotted on the left y-axis in seconds.

4.4.2 Low-dimensional data

We now present the performance of J_{RSJ}, J_{EGO}, J_{EGO*} and J_G for various settings. The cost of building indexes for J_{RSJ} is ignored, giving it an advantage.

The x-axis plots the values of ε, which are varied so that meaningful selectivity is achieved. Clearly, if selectivity is 0, then ε is too small and vice versa if the selectivity is more than 100.

In all but one graph the left y-axis represents the total time in seconds to do the join for the given settings. The right y-axis plots the selectivity values for each value of ε in the experiments, in actual number of matching points. As expected, in each graph the selectivity, shown by the line with the '$\times$', increases as ε increases.

J_{RSJ} is depicted only in Figures 4.14 and 4.15 because for all tested cases it has

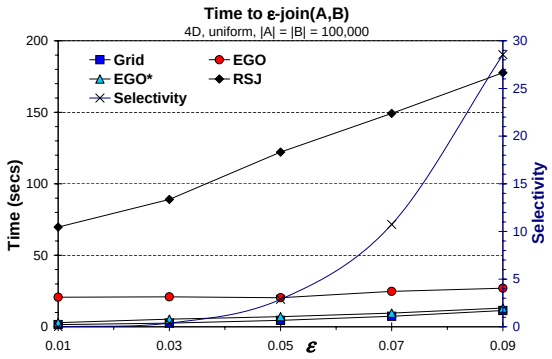


Figure 4.14 Join 4D uniform data

shown much worse results than the other joins, Figure 4.14 depicts performance of the joins for 4-dimensional uniform data with cardinality of both sets being 10^5. Figure 4.15 shows the performance of the same joins relative to that of J_{RSJ}.

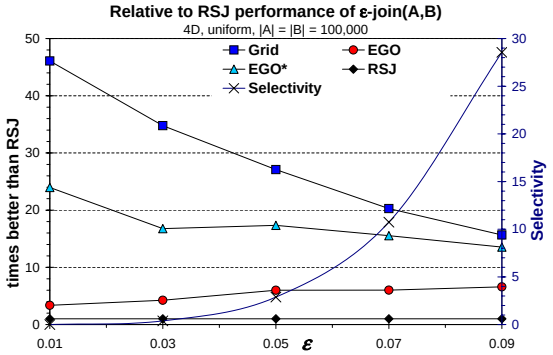


Figure 4.15 Join 4D uniform data, cost relative to J_{RSJ}

In Figure 4.15, J_{EGO} shows 3.5–6.5 times better results than those of J_{RSJ}, which corroborates the fact that, by itself, J_{EGO} is a quite competitive scheme for low-dimensional data. But it is not as good as the two new schemes.

Next comes J_{EGO*} whose performance is *always* better than that of J_{EGO} in all experiments. This shows the strength of J_{EGO*}. Because of the selectivity, the values of ε are likely to be small for low-dimensional data and large for high-dimensional data. The EGO-heuristic is not well-suited for small values of ε. The smaller the epsilon, the less likely that a sequence has an inactive dimension. In Figure 4.15 J_{EGO*} is seen to give 13.5–24 times better performance than J_{RSJ}.

Another trend that can be observed from the graphs is that J_G is better that J_{EGO*}, except for high-selectivity cases (Figure 4.19). J_{EGO} shows results several times worse than those of J_G, which corroborates the choice of the Grid-index which also was the clear winner in our comparison with main memory optimized versions of R-tree, R*-tree, CR-tree, and quad-tree indexes in Chapter 3. In Figure 4.15 J_G showed 15.5–46 times better performance than J_{RSJ}.

Unlike J_{EGO}, J_{EGO*} always shows results at least comparable to those of J_G. For all the methods, the difference in relative performance shrinks as ε (and selectivity) increases.

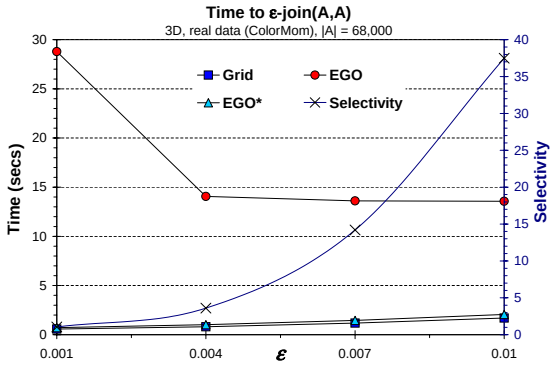


Figure 4.16 Join 3D real data

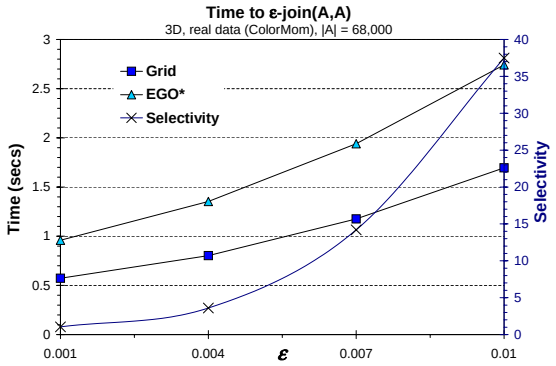


Figure 4.17 Join 3D real data without J_{EGO} (for clarity)

Figures 4.16 and 4.17 show the results for the self-join of real 3-dimensional data taken from the ColorMom file. The cardinality of the set is 68,000. The graph on the left shows the best three schemes, and the graph on the right omits J_{EGO} scheme due to its much poorer performance. From these two graphs we can see that J_G is almost two times better than J_{EGO*} for small values of ε.

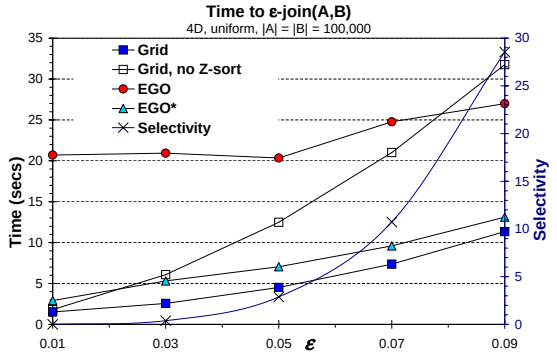


Figure 4.18 Join 4D uniform data $|A| = |B| = 100,000$

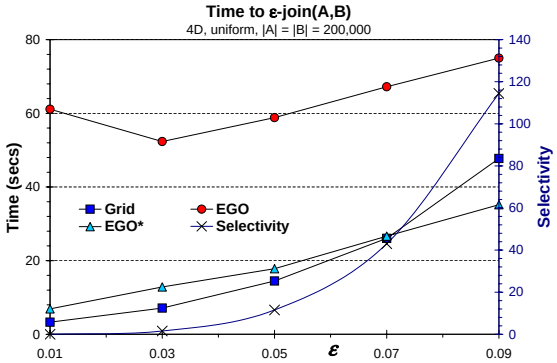


Figure 4.19 Join 4D uniform data $|A| = |B| = 200,000$

Figures 4.18 and 4.19 show the results for 4-dimensional uniform data. The graph on the left is for sets of cardinality 100,000, and that on the right is for sets with cardinality 200,000. Figure 4.18 emphasizes the importance of performing Z-sort on data being joined: the performance improvement is ~ 2.5 times. J_G without Z-sort, in general, while being better than J_{EGO}, shows worse results than that of J_{EGO*}.

Figure 4.19 presents another trend. In this figure J_{EGO*} becomes a better choice than J_G for values of ε greater than ~ 0.07. This choice of epsilon corresponds to a high selectivity of ~ 43. Therefore J_{EGO*} can be applied for joining high selectivity cases for low-dimensional data.

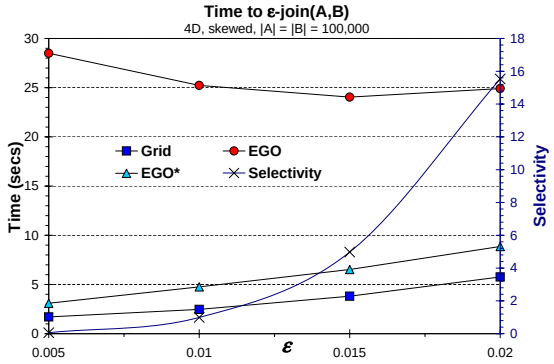


Figure 4.20 Join 4D skewed data

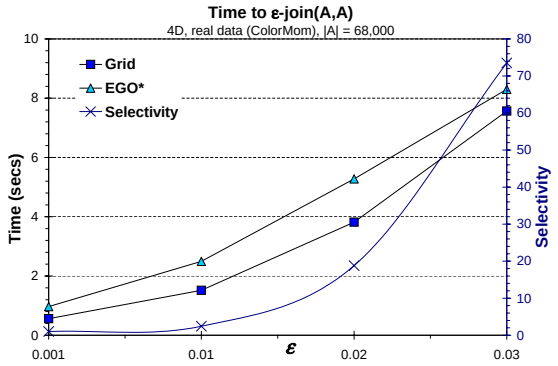


Figure 4.21 Join 4D real data

Figures 4.20 and 4.21 show the results for 4-dimensional skewed and real data. Note that the values of ε are now varied over a smaller range than that of the uniformly distributed case. This is so because in these cases points are closer together and smaller values of ε are needed to achieve the same selectivity as in uniform case. In these graphs J_{EGO}, J_{EGO*}, and J_G exhibit behavior similar to that in the previous figures with J_G being the best scheme.

4.4.3 High-dimensional data

We now study the performance of the various algorithms for higher dimensions. Figures 4.22 and 4.23 show the results for 9-dimensional data for uniformly distributed

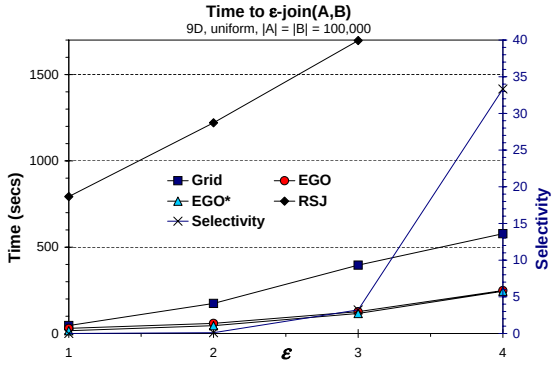


Figure 4.22 Join 9D uniform data

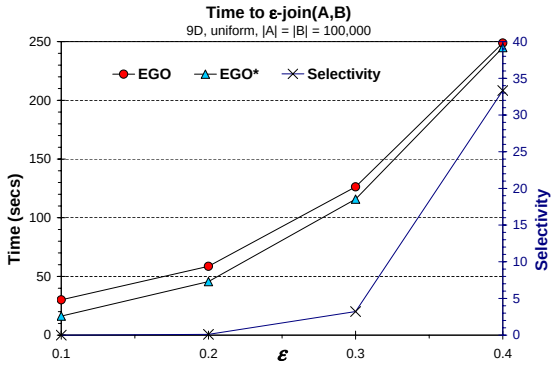


Figure 4.23 Join 9D uniform data, the best two techniques

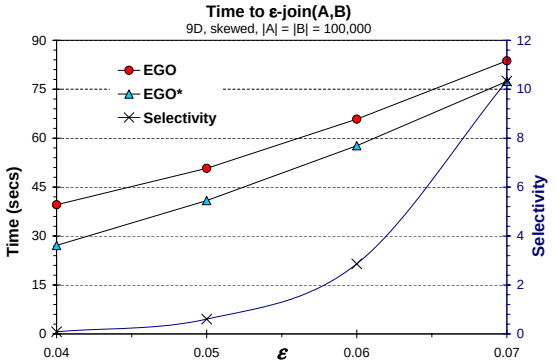


Figure 4.24 Join 9D skewed data

data. Figure 4.24 presents the results for 9-dimensional skewed data, Figure 4.25 gives the results for real 9-dimensional data. Figures 4.26 and 4.27 show the results with

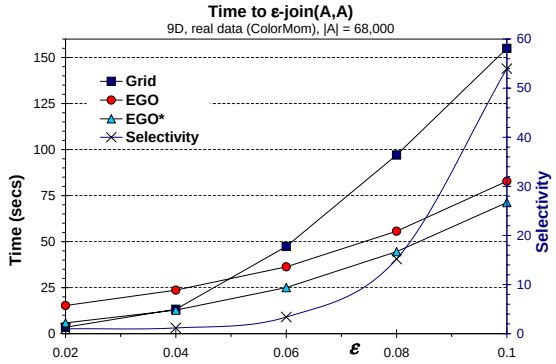


Figure 4.25 Join 9D real data

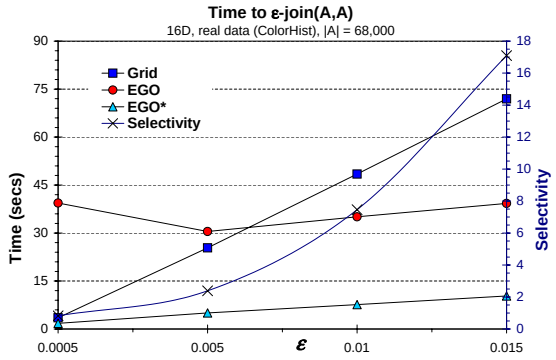


Figure 4.26 Join 16D real data

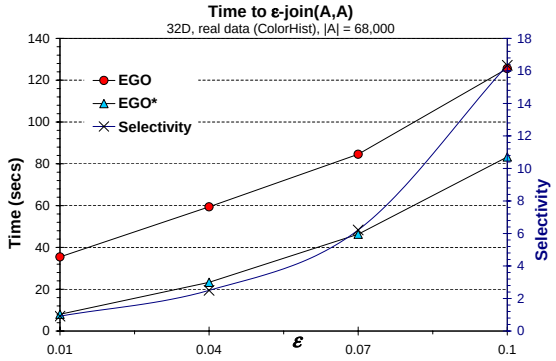


Figure 4.27 Join 32D real data

the 16- and 32-dimensional real data respectively. As with low-dimensional data, for

all tested cases, J_{RSJ} had the worst results. Therefore, the performance of J_{RSJ} is omitted from most graphs – only one representative case is shown in Figure 4.22.

An interesting change in the relative performance of J_G is observed for high-dimensional data. Unlike the case of low-dimensional data, J_{EGO} and J_{EGO*} give better results than J_G. J_G is not competitive for high-dimensional data, and its results are often omitted for clear presentation of J_{EGO} and J_{EGO*} results. A consistent trend in all graphs is that J_{EGO*} results are *always* better than those of J_{EGO} . The difference is especially noticeable for the values of ε corresponding to low selectivity. This is a general trend: J_{EGO} does not work well for smaller epsilons, because in this case a sequences is less likely to have an inactive dimension. J_{EGO*} does not suffer from this limitation.

Set Cardinality When the join of two sets is to be computed using Grid-join, an index is built on one of the two sets. Naturally, the question of which set to build the index on arises. We ran experiments to study this issue. The results indicate that building the index on the smaller dataset always gave better results.

4.5 Related work

The problem of the spatial join of two datasets is to identify pairs of objects, one from each dataset, such that they satisfy a certain constraint. If both datasets are the same, this corresponds to a self-join. The most common join constraint is that of proximity: i.e. the two objects should be within a certain distance of each other. This corresponds to the ε-join where ε is the threshold distance beyond which objects are no longer considered close enough to be joined. Below we discuss some of the most prominent solutions for efficient computation of similarity joins.

Shim et. al. [SSA97] propose to use ε-KDB-tree for performing high-dimensional similarity joins of massive data. The main-memory based ε-KDB-tree and the corresponding algorithm for similarity join are modified to produce a disk-based solution

that can scale to larger datasets. Whenever the number of points in a leaf node exceed a certain threshold it is split into $\lfloor 1/\varepsilon \rfloor$ stripes[4] each of width equal to or slightly greater than ε in the i^{th} dimension. If the leaf node is at level i, then the i^{th} dimension is used for splitting. The join is performed by traversing the index structures for each of the data sets. Each leaf node can join only with its two adjacent siblings. The points are first sorted with the first splitting dimension and stored in an external file.

The R-Tree Spatial Join (RSJ) algorithm [BKS93] works with an R-tree index built on the two datasets being joined. The algorithm is recursively applied to corresponding children if their minimum bounding rectangles (MBRs) are within distance ε of each other. Several optimizations of this basic algorithm have been proposed [HJR97]. A cost model for spatial joins was introduced in [BK01]. The Multipage Index (MuX) was also introduced that optimizes for I/O and CPU cost at the same time.

In [PD96] Patel et. al a plane sweeping technique is modified to create a disk-based similarity join for 2-dimensional data. The new procedure is called the Partition Based Spatial Merge join, or PBSM-join. A partition based merge join is also presented in [LR96]. Shafer et al in [SA97] present a method of parallelizing high-dimensional proximity joins. The ε-KDB-tree is parallelized and compared with the approach of space partitioning. Koudas et al [KS98] have proposed a generalization of the Size Separation Spatial Join Algorithm, named Multidimensional Spatial Join (MSJ).

Recently, Böhm et al [BBKK01] proposed the EGO-join. Both sets of points being joined are first sorted in accordance with the so called Epsilon Grid Order (EGO). The EGO-join procedure is recursive. A heuristic is utilized for determining non-joinable sequences. More details about EGO-join will be covered in Section 4.2.1. The EGO-join was shown to outperform other join methods in [BBKK01].

A excellent review of multidimensional index structures including grid-like and Quad-tree based structures can be found in [TUW98]. Main-memory optimization of disk-based index structures has been explored recently for B+-trees [RR00] and

[4]Note that for high-dimensional data ε can easily exceed 0.5 rendering this approach into a brute force method.

multidimensional indexes [KCK01]. Both studies investigate the redesign of the nodes in order to improve cache performance.

4.6 Conclusions

Table 4.2 Choice of Join Algorithm

	Small ε	**Average ε**	**Large ε**
Low Dim	J_G	J_G	J_G or J_{EGO*}
High Dim	J_G or J_{EGO*}	J_{EGO*}	J_{EGO*}

In this chapter we considered the problem of similarity join in main memory for low- and high-dimensional data. We propose two new algorithms: *Grid-join* and *EGO*-join* that were shown to give superior performance than the state-of-the-art technique (EGO-join) and RSJ.

The significance of the choice of ε and recommendations for a good choice for testing and comparing algorithms with meaningful selectivity were discussed. We demonstrated an example with values of ε too small for the given dimensionality where one methods showed the best results over the others whereas with more meaningful settings it would show the worst results.

While recent research has concentrated on joining high-dimensional data, little attention was been given to the choice of technique for low-dimensional data. In our experiments, the proposed Grid-join approach showed the best results for low-dimensional case or when values of ε are very small. The EGO*-join has demonstrated substantial improvement over EGO-join for all the cases considered and is the best choice for high-dimensional data or when values of ε are large. The results of the experiments with RSJ proves the strength of Grid-join and EGO*-join.

An analytical study has been presented for selecting the grid size. As a side effect of the study the cost-estimating function for the Grid-join has been developed. This function can be used by a query optimizer for selecting the best execution plan.

Based upon the experimental results, the recommendation for choice of join algorithm is summarized in Table 4.6.

5. PROBABILISTIC QUERIES OVER IMPRECISE DATA

5.1 Introduction

In many applications, sensors are used to continuously track or monitor the status of an environment. Readings from the sensors are sent back to the application, and decisions are made based on these readings. For example, temperature sensors installed in a building are used by a central air-conditioning system to decide if the temperature of any room needs to be adjusted or to detect other problems. Sensors distributed in the environment can be used to detect if hazardous materials are present and how they are spreading. In a moving object database, objects are constantly monitored and a central database may collect their updated locations.

The framework for many of these applications includes a database or server to which the readings obtained by the sensors or the locations of the moving objects are sent. Users query this database to find information of interest. Due to several factors such as limited network bandwidth to the server and limited battery power of the mobile device, it is often infeasible for the database to contain the exact status of an entity being monitored at every moment in time. An inherent property of these applications is that readings from sensors are sent to the central server periodically. In particular, at any given point in time, the recorded sensor reading is likely to be different from the actual value. The correct value of a sensor's reading is known only when an update is received. Under these conditions, the data in the database is only an estimate of the actual state of the environment at most points in time.

This inherent uncertainty of data affects the accuracy of answers to queries. Figure 5.1(a) illustrates a query that determines the sensor (either x or y) that reports the lower temperature reading. Based upon the data available in the database (x_0 and y_0), the query returns "x" as the result. In reality, the temperature readings

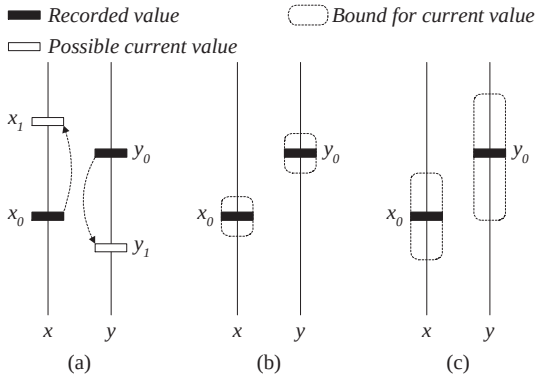


Figure 5.1 Example of sensor data and uncertainty.

could have changed to values x_1 and y_1, in which case "y" is the correct answer. This example demonstrates that the database does not always truly capture the state of the external world, and the value of the sensor readings can change without being recognized by the database. Sistla et. al. [SWCD98] identify this type of data as a *dynamic attribute*, whose value changes over time even if it is not explicitly updated in the database. In this example, because the exact values of the data items are not known to the database between successive updates, the database incorrectly assumes that the recorded value is the actual value and produces incorrect results.

Given the uncertainty of the data, providing meaningful answers seems to be a futile exercise. However, one can argue that in many applications, the values of objects cannot change drastically in a short period of time; instead, the degree and/or rate of change of an object may be constrained. For example, the temperature recorded by a sensor may not change by more than a degree in 5 minutes. Such information can help solve the problem. Consider the above example again. Suppose we can provide a guarantee that at the time the query is evaluated, the actual values monitored by x and y could be no more than some deviations, d_x and d_y, from x_0 and y_0, respectively, as shown in Figure 5.1(b). With this information, we can state with confidence that x yields the minimum value.

In general, the uncertainty of the objects may not allow us to identify a single object that has the minimum value. For example, in Figure 5.1(c), both x and y have the possibility of recording the minimum value since the reading of x may not be lower than that of y. A similar situation exists for other types of queries such as those that request a numerical value (e.g., "What is the lowest temperature reading?"). For these queries too, providing a single value may be infeasible due to the uncertainty in each object's value. Instead of providing a definite answer, the database can associate different levels of confidence with each answer (e.g., as a probability) based upon the uncertainty of the queried objects.

The notion of probabilistic answers to queries over uncertain data has not been well studied. Wolfson et. al briefly touched upon this idea [WSCY99] for the case of range queries in the context of a moving object database. The objects are assumed to move in straight lines with a known average speed. The answers to the queries consist of objects' identities and the probability that each object is located in the specified range. In this chapter we extend the notion of probabilistic queries to cover a much broader class of queries. The class of queries considered includes aggregate queries that compute answers over a number of objects. We also discuss the importance of the nature of answer requested by a query (identity of object versus the value). For example, we show that there is a significant difference between the following two queries: "Which object has the minimum temperature?" versus "What is the minimum temperature?". Furthermore, we relax the model of uncertainty so that any reasonable model can be used by the application. Our techniques are applicable to the common models of uncertainty that have been proposed elsewhere.

The probabilities in the answer allow the user to place appropriate confidence in the answer as opposed to having an incorrect answer or no answer at all. Depending upon the application, one may choose to report only the object with the highest probability as having the minimum value, or only those objects whose probability exceeds a minimum probability threshold. Our proposed work will be able to work with any of these models.

Answering aggregate queries (such as *minimum* or *average*) is much more challenging than range queries, especially in the presence of uncertainty. The answer to a probabilistic range query consists of a set of objects along with a non-zero probability that the object lies in the query range. Each object's probability is determined by the uncertainty of the object's value and the query range. However, for aggregate queries, the interplay between multiple objects is critical. The resulting probabilities are greatly influenced by the uncertainty of attribute values of other objects. For example, in Figure 5.1(c) the probability that x has the minimum value is affected by the relative value and bounds for y.

A probabilistic answer also reflects a certain level of uncertainty that results from the uncertainty of the queries object values. If the uncertainty of all (or some) of the objects was reduced (or eliminated completely), the uncertainty of the result improves. For example, without any knowledge about the value of an object, one could arguably state that it falls within a query range with 50% probability. On the other hand, if the value is known perfectly, one can state with 100% confidence that the object either falls within or outside the query range. Thus the quality of the result is measured by the degree of ambiguity in the answer. We therefore need metrics to evaluate the quality of a probabilistic answer. We propose metrics for evaluating the quality of the probabilistic answers in this chapter. As we shall see, it turns out that different metrics are suitable for different classes of queries.

It is possible that the quality of a query result may not be acceptable for certain applications – a more definite result may be desirable. Since the poor quality is directly related to the uncertainty in the object values, one possibility for improving the results is to delay the query until the quality improves. However this is an unreasonable solution due to the increased query response time. Instead, the database could request updates from all objects (e.g., sensors) – this solution incurs a heavy load on the resources. In this chapter, we propose to request updates only from objects that are being queried, and furthermore those that are likely to improve the quality of the query result. We present a number of heuristics and an experimental

evaluation. These policies attempt to optimize the use of the constrained resource (e.g., network bandwidth to the server) to improve average query quality.

It should be noted that the imprecision in the query answers is inherent in this problem (due to uncertainty in the actual values of the dynamic attribute), in contrast to the problem of providing approximate answers for improved performance wherein accuracy is traded for efficiency.

To sum up, the contributions introduced in this chapter are:

- A broad classification of probabilistic queries over uncertain data, based upon a flexible model of uncertainty;
- Techniques for evaluating probabilistic queries, including optimizations;
- Metrics for quantifying the quality of answers to probabilistic queries;
- Policies for improving the quality of answers to probabilistic queries under resource constraints.

The material presented in this chapter is a joint work with Rey Cheng.

The rest of this chapter is organized as follows. In Section 5.2 we describe a general model of uncertainty, and the concept of probabilistic queries. Sections 5.3 and 5.4 discuss the algorithms for evaluating different kinds of probabilistic queries. Section 5.5 discusses quality metrics that are appropriate to these queries. Section 5.6 proposes update policies that improve the query answer quality. We present an experimental evaluation of the effectiveness of these update policies in Section 5.7. Section 5.8 discusses related work and Section 5.9 concludes the chapter.

5.2 Probabilistic queries

In this section, we describe the model of uncertainty considered in this chapter. This is a generic model, as it can accommodate a large number of application paradigms. Based on this model, we introduce a number of probabilistic queries.

5.2.1 Uncertainty model

One popular model for uncertainty for a dynamic attribute is that at any point in time, the actual value is within a certain bound d of its last reported value. If the actual value changes more than d, then the sensor reports its new value to the database and possibly changes d. For example, [WSCY99] describes a moving-object model where the location of an object is a dynamic attribute, and an object reports its location to the server if its deviation from the last reported location exceeds a threshold. Another model assumes that the attribute value changes with known speed, but the speed may change each time the value is reported. Other models include those that have no uncertainty. For example, in [PJ], the exact speed and direction of movement of each object are known. This model requires updates at the server whenever an object's speed or direction change.

For the purpose of our discussion, the exact model of uncertainty is unimportant. All that is required is that at the time of query execution the range of possible values of the attribute of interest are known. We are interested in queries over some dynamic attribute a of a set of database objects T. Also, we assume that a is a real-valued attribute, although our models and algorithms can be extended to other domains easily. We denote the ith object of T by T_i and the value of attribute a of T_i by $T_i.a$ (where $i = 1, \ldots, |T|$). Throughout this chapter, we treat $T_i.a$ at any time instant t as a continuous random variable. Sometimes instead of writing $T_i.a(t)$ meaning the value of the attribute at time instant t we write just $T_i.a$ for short. The uncertainty of $T_i.a(t)$ can be characterized by the following two definitions (we use *pdf* to abbreviate the term "probability density function"):

Definition 5.1: An **uncertainty interval** of $T_i.a(t)$, denoted by $U_i(t)$, is an interval $[l_i(t), u_i(t)]$ such that $l_i(t)$ and $u_i(t)$ are real-valued functions of t, $l_i(t) \leq u_i(t)$, and that the conditions $u_i(t) \geq l_i(t)$ and $T_i.a(t) \in [l_i(t), u_i(t)]$ always hold.

Definition 5.2: An **uncertainty pdf** of $T_i.a(t)$, denoted by $f_i(x,t)$, is a pdf of $T_i.a(t)$ such that $f_i(x,t) = 0$ for $\forall x \notin U_i(t)$.

Notice that since $f_i(x,t)$ is a pdf, it has the property that $\int_{l_i(t)}^{u_i(t)} f_i(x,t)dx = 1$. The above definition specifies the uncertainty of $T_i.a(t)$ in terms of a closed interval and the probability distribution of $T_i.a(t)$. Notice that this definition does not specify how the uncertainty interval evolves over time, and what the pdf $f_i(x,t)$ is inside the uncertainty interval. The only requirement for $f_i(x,t)$ is that its value is 0 outside the uncertainty interval. Usually, the scope of uncertainty is determined by the last recorded value, the time elapsed since its last update, and some application-specific assumptions. For example, one may decide that $U_i(t)$ contains all the values within a distance of $(t - t_{update}) \times v$ from its last reported value, where t_{update} is the time that the last update was obtained, and v is the maximum rate of change of the value. One may also specify that $T_i.a(t)$ is uniformly distributed inside the interval, i.e., $f_i(x,t) = 1/[u_i(t) - l_i(t)]$ for $\forall x \in U_i(t)$, assuming that $u_i(t) > l_i(t)$. Note that the uniform distribution represents the worst-case uncertainty over a given interval.

5.2.2 Classification of probabilistic queries

We now present a classification of probabilistic queries and examples of common representative queries for each class. We identify two important dimensions for classifying database queries. First, queries can be classified according to the nature of the answers. An *entity-based* query returns a set of objects (e.g., sensors) that satisfy the condition of the query. A *value-based* query returns a single value, examples of which include querying the value of a particular sensor, and computing the average value of a subset of sensor readings. The second property for classifying queries is whether aggregation is involved. We use the term aggregation loosely to refer to queries where interplay between objects determines the result. In the following definitions, we use the following naming convention: the first letter is either E (for entity-based queries) or V (for value-based queries).

1. Value-based Non-Aggregate Class

This is the simplest type of query in our discussions. It returns an attribute value of an object as the only answer, and involves no aggregate operators. One example of a probabilistic query for this class is the *VSingleQ*:

Definition 5.3: **Probabilistic Single Value Query (VSingleQ)** When querying $T_k.a(t)$, a VSingleQ returns bounds l and u of an uncertainty region of $T_k.a(t)$ and its pdf $f_k(x,t)$.

An example of VSingleQ is "What is the wind speed recorded by sensor s_{22}?" Observe how this definition expresses the answer in terms of a bounded probabilistic value, instead of a single value. Also notice that $\int_{l_i(t)}^{u_i(t)} f_i(x,t)\,dx = 1$.

2. Entity-based Non-Aggregate Class

This type of query returns a set of objects, each of which satisfies the condition(s) of the query, independent of other objects. A typical example of this class is the ERQ, defined below.

Definition 5.4: **Probabilistic Range Query (ERQ)** Given at time instant t a closed interval $[l,u]$, where $l,u \in \Re$ and $l \leq u$, an ERQ returns set R of all tuples (T_i,p_i), where p_i is the non-zero probability that $T_i.a(t) \in [l,u]$, i.e. $R = \{(T_i,p_i) : p_i = \mathrm{P}(T_i.a(t) \in [l,u]) \text{ and } p_i > 0\}$

An *ERQ* returns a set of objects, augmented with probabilities, that satisfy the query interval.

3. Entity-based Aggregate Class

The third class of query returns a set of objects which satisfy an aggregate condition. We present the definitions of three typical queries for this class. The first two return objects with the minimum or maximum value of $T_i.a$:

Definition 5.5: **Probabilistic Minimum (Maximum) Query (EMinQ (EMaxQ))** An EMinQ (EMaxQ) returns set R of all tuples (T_i, p_i), where p_i is the non-zero probability that $T_i.a$ is the minimum (maximum) value of a among all objects in T.

A one-dimensional nearest neighbor query, which returns object(s) having a minimum absolute difference of $T_i.a$ and a given value q, is also defined:

Definition 5.6: **Probabilistic Nearest Neighbor Query (ENNQ)** Given a value $q \in \Re$, an ENNQ returns set R of all tuples (T_i, p_i), where p_i is the non-zero probability that $|T_i.a - q|$ is the minimum among all objects in T.

Note that for all the queries we defined in this class the condition $\sum_{T_i \in R} p_i = 1$ holds.

4. Value-based Aggregate Class

The final class involves aggregate operators that return a single value. Example queries for this class include:

Definition 5.7: **Probabilistic Average (Sum) Query (VAvgQ (VSumQ))** A VAvgQ (VSumQ) returns bounds l and u of an uncertainty interval and pdf $f_X(x)$ of r.v. X representing the average (sum) of the values of a for all objects in T.

Definition 5.8: **Probabilistic Minimum (Maximum) Value Query (VMinQ (VMaxQ))** A VMinQ (VMaxQ) returns bounds l and u of an uncertainty interval and pdf $f_X(x)$ of r.v. X representing the minimum (maximum) value of a among all objects in T.

All these aggregate queries return answers in the form of a pdf $f_X(x)$ and a closed interval $[l, u]$, such that $\int_l^u f_X(x)\,dx = 1$.

Table 5.2.2 summarizes the basic properties of the probabilistic queries discussed above. For illustrating the difference between probabilistic and non-probabilistic

Table 5.1 Classification of Probabilistic Queries.

Query Class	**Entity-based**	**Value-based**
Aggregate	ENNQ, EMinQ, EMaxQ	VAvgQ, VSumQ, VMinQ, VMaxQ
Non-Aggr.	ERQ	VSingleQ
Answer (Probabilistic)	$\{(T_i, p_i) : 1 \leq i \leq \|T\| \wedge p_i > 0\}$	$l, u, f_X(x)$
Answer (Non-Prob.)	$\{T_i : 1 \leq i \leq \|T\|\}$	$x \in \Re$

queries, the last row of the table lists the forms of answers expected if probability information is not augmented to the result of the queries e.g., the non-probabilistic version of EMaxQ is a query that returns object(s) with maximum values based only on the recorded values of $T_i.a$. It can be seen that the probabilistic queries provide more information on the answers than their non-probabilistic counterparts.

Example. We illustrate the properties of the probabilistic queries with a simple example. In Figure 5.2, readings of four sensors s_1, s_2, s_3 and s_4, each with a different uncertainty interval, are being queried at time t. Assume that readings of these sensors $s_1(t)$, $s_2(t)$, $s_3(t)$, and $s_4(t)$ are uniformly distributed on their uncertainty intervals $[l_1, u_1]$, $[l_2, u_2]$, $[l_3, u_3]$ and $[l_4, u_4]$. A VSingleQ applied on s_4 at time t gives us the result: l_4, u_4, $f_{s_4}(x) = 1/(u_4 - l_4)$. When an ERQ (represented by the interval $[l, u]$) is invoked at time t to find out how likely each reading is inside $[l, u]$, we see that the reading of s_1 is always inside the interval. It therefore has a probability of 1 for satisfying the ERQ. The reading of s_4 is always outside the rectangle, thus it has

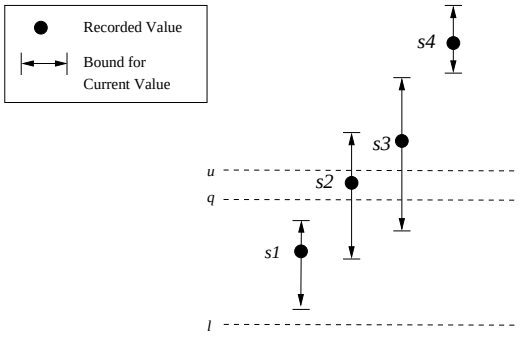


Figure 5.2 Illustrating the probabilistic queries

a probability of 0 of being located inside $[l, u]$. Since $U_2(t)$ and $U_3(t)$ partially overlap $[l, u]$, s_2 and s_3 have some chance of satisfying the query. In this example, the result of the ERQ is: $\{(s_1, 1), (s_2, 0.7), (s_3, 0.4)\}$.

In the same figure, an EMinQ is issued at time t. We observe that s_1 has a high probability of having the minimum value, because a large portion of the $U_1(t)$ has a smaller value than the others. The reading of s_1 has a high chance of being located in this portion because $s_1(t)$ has the uniform distribution. The reading of s_4 does not have any chance of yielding the minimum value, since none of the values inside $U_4(t)$ is smaller than others. The result of the EMinQ for this example is: $\{(s_1, 0.7), (s_2, 0.2), (s_3, 0.1)\}$. On the other hand, an EMaxQ will return $\{(s_4, 1)\}$ as the only result, since every value in $U_4(t)$ is larger than any readings from the other sensors, and we are assured that s_4 yields the maximum value. An ENNQ with a query value q is also shown, where the results are: $\{(s_1, 0.2), (s_2, 0.5), (s_3, 0.3)\}$.

When a value-based aggregate query is applied to the scenario in Figure 5.2, a bounded pdf $p(x)$ is returned. If a VSumQ is issued, the result is a distribution in $[l_1{+}l_2{+}l_3{+}l_4, u_1{+}u_2{+}u_3{+}u_4]$; each x in this interval is the sum of the readings from the four sensors. The result of a VAvgQ is a pdf in $[(l_1{+}l_2{+}l_3{+}l_4)/4, (u_1{+}u_2{+}u_3{+}u_4)/4]$. The results of VMinQ and VMaxQ are probability distributions in $[l_1, u_1]$ and $[l_4, u_4]$ respectively, since only the values in these ranges have a non-zero probability value of satisfying the queries.

5.3 Evaluating entity-based queries

In this section we examine how the probabilistic entity-based queries introduced in the last section can be answered. We start with the discussion of an ERQ, followed by a more complex algorithm for answering an ENNQ. We also show how the algorithm for answering an ENNQ can be easily changed for EMinQ and EMaxQ.

5.3.1 Evaluation of ERQ

Recall that ERQ returns a set of tuples (T_i, p_i) where p_i is the non-zero probability that $T_i.a$ is within a given interval $[l, u]$. Let R be the set of tuples returned by the ERQ. The algorithm for evaluating the ERQ at time instant t is described in Figure 5.3.

---

1. $R \leftarrow \emptyset$
2. **for** $i \leftarrow 1$ **to** $|T|$ **do**
   (a) $D \leftarrow U_i(t) \cap [l, u]$
   (b) **if** $(D \neq \emptyset)$ **then**
      i. $p_i \leftarrow \int_D f_i(x, t)\, dx$
      ii. **if** $p_i \neq 0$ **then** $R \leftarrow R \cup \{(T_i, p_i)\}$
3. **return** R

Figure 5.3 ERQ Algorithm.

In this algorithm, each object in T is checked once. To evaluate p_i for T_i, we first compute the overlapping interval D of the two intervals: $U_i(t)$ and $[l, u]$ (Step 2a). If D is the empty set, we are assured that $T_i.a$ does not lie in $[l, u]$, and by the definition of ERQ, T_i is not included in the result. Otherwise, we calculate the probability that

$T_i.a$ is inside $[l, u]$ by integrating $f_i(x, t)$ over D, and put the result into R if $p_i \neq 0$ (Step 2b). The set of tuples (T_i, p_i), stored in R, are returned in Step 3.

5.3.2 Evaluation of ENNQ

Processing an ENNQ involves evaluating the probability of the attribute a of each object T_i being the closest to (nearest-neighbor of) a value q. In general, this query can be applied to multiple attributes, such as coordinates. In particular, it could be a nearest-neighbor query for moving objects. Unlike the case of ERQ, we can no longer determine the probability for a object independent of the other objects. Recall that an ENNQ returns a set of tuples (T_i, p_i) where p_i denotes the non-zero probability that T_i has the minimum value of $|T_i.a - q|$. Let S be the set of objects to be considered by the ENNQ, and let R be the set of tuples returned by the query. The algorithm presented here consists of 4 phases: *projection*, *pruning*, *bounding* and

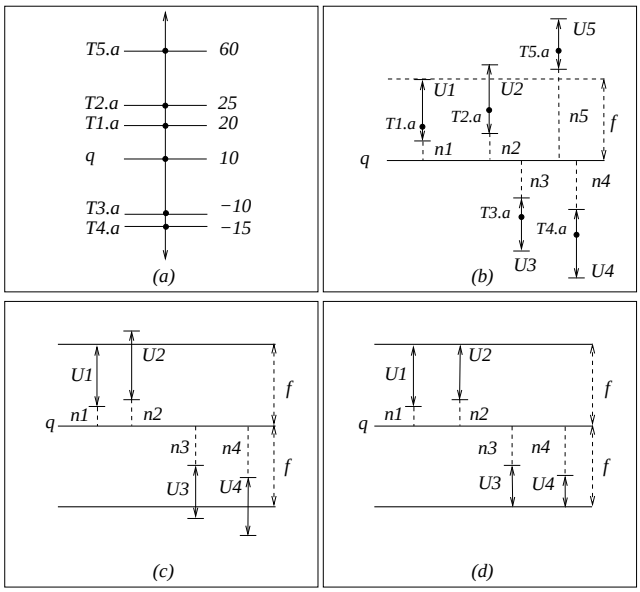


Figure 5.4 Phases of the ENNQ algorithm.

evaluation. The first three phases filter out objects in T whose values of a have no

chance of being the closest to q. The final phase, *evaluation*, is the core of our solution: for every object T_i that remains after the first three phases, the probability that $T_i.a$ is nearest to q is computed.

1. **Projection Phase.**

In this phase, the uncertainty interval of each $T_i.a$ is computed based on the uncertainty model used by the application. Figure 5.4(a) shows the last recorded values of $T_i.a$ in S at time t_0, and the uncertainty intervals are shown in Figure 5.4(b).

2. **Pruning Phase.**

Consider two uncertainty intervals $U_1(t)$ and $U_2(t)$. If the smallest distance between $U_1(t)$ and q is larger than the largest distance between $U_2(t)$ and q, we can immediately conclude that T_1 is not an answer to the ENNQ: even if the actual value of $T_2.a$ is as far as possible from q, $T_1.a$ still has no chance to be closer to q than $T_2.a$. Based on this observation, we can eliminate objects from T by the algorithm shown in Figure 5.5. In this algorithm, N_i and F_i record the closest and farthest possible values of $T_i.a$ to q, respectively. Steps 1(a) to (d) assign proper values to N_i and F_i. If q is inside the interval $U_i(t)$, then N_i is taken as the point q itself. Otherwise, N_i is either $l_i(t)$ or $u_i(t)$, depending on which value is closer to q. F_i is assigned in a similar manner. After this phase, S contains the (possibly fewer) objects which must be considered by q. This is the minimal set of objects which must be considered by the query since any of them can have a value of $T_i.a$ closest to q. Figure 5.4(b) illustrates how this phase removes T_5, which is irrelevant to the ENNQ, from S.

3. **Bounding Phase.**

For each object in S, there is no need to examine all portions in its uncertainty interval. We only need to look at the regions that are located no farther than f from q. We do this conceptually by drawing a *bounding interval* B of length $2f$, centered at q. Any portion of the uncertainty interval outside B can be ignored. Figure 5.4(c) shows a bounding interval with length $2f$, and (d) illustrate the result of this phase.

1. **for** $i \leftarrow 1$ **to** $|S|$ **do**
   (a) **if** $q \in U_i(t)$ **then** $N_i \leftarrow q$
   (b) **else**
      i. **if** $|q - l_i(t)| < |q - u_i(t)|$ **then** $N_i \leftarrow l_i(t)$
      ii. **else** $N_i \leftarrow u_i(t)$
   (c) **if** $|q - l_i(t)| < |q - u_i(t)|$ **then** $F_i \leftarrow u_i(t)$
   (d) **else** $F_i \leftarrow l_i(t)$
2. $f \leftarrow min_{1 \leq i \leq |S|}|F_i - q|$; $m \leftarrow |S|$
3. **for** $i \leftarrow 1$ **to** m **do**
   if $(|N_i - q| > f)$ **then** $S \leftarrow S \setminus \{T_i\}$
4. **return** S

Figure 5.5 Algorithm for the Pruning Phase of ENNQ.

The phases we have just described attempt to reduce the number of objects to be evaluated, and derive an upper bound on the range of values to be considered.

4. **Evaluation Phase.**

Based on S and the bounding interval B, our aim is to calculate, for each object in S, the probability that it is the nearest neighbor of q. In the *pruning phase*, we have already found N_i, the point in $U_i(t)$ nearest to q. Let us call $|N_i - q|$ the *near_distance* of T_i, or n_i. Let as define new r.v. X_i such that $X_i = |T_i.a(t) - q|$. Also, let $F_i(x)$ be the X_i's cdf, i.e., $F_i(x) = \mathrm{P}(|T_i.a(t) - q| \leq x)$, and $f_i(x)$ be its pdf. Figure 5.6 presents the algorithm for this phase.

Note that if $T_i.a(t)$ has no uncertainty i.e., $U_i(t)$ is exactly equal to $T_i.a(t)$, the evaluation phase algorithm needs to be modified. In the rest of this section, we will explain how the evaluation phase works, assuming non-zero uncertainty.

1. $R \leftarrow \emptyset$
2. Sort the elements in S in ascending order of n_i, and
   rename the sorted elements in S as $T_1, T_2, \ldots, T_{|S|}$
3. $n_{|S|+1} \leftarrow f$
4. **for** $i \leftarrow 1$ **to** $|S|$ **do**
   (a) $p_i \leftarrow 0$
   (b) **for** $j \leftarrow i$ **to** $|S|$ **do**
      i. $p \leftarrow \int_{n_j}^{n_{j+1}} f_i(x) \cdot \prod_{k=1 \wedge k \neq i}^{j} (1 - F_k(x))\, dx$
      ii. $p_i \leftarrow p_i + p$
   (c) $R \leftarrow R \cup \{(T_i, p_i)\}$
5. **return** R

Figure 5.6 Algorithm for the Evaluation Phase of ENNQ.

Evaluation of $F_i(x)$ and $f_i(x)$ To understand how the evaluation phase works, it is crucial to know how to obtain $F_i(x)$. As introduced before, $F_i(x)$ is the cdf of X_i, and thus $F_i(x) \stackrel{def}{=} \mathrm{P}(|T_i.a(t) - q| \leq x)$. We illustrate the evaluation of $F_i(x)$ in Figure 5.7.

1. **if** $x < n_i$ **return** 0
2. **if** $x > |q - F_i|$, **return** 1
3. $D \leftarrow U_i(t) \cap [q - x, q + x]$
4. **return** $\int_D f_i(x, t)\, dx$

Figure 5.7 Computation of $F_i(x)$.

Recall that $f_i(x)$ is the pdf of X_i and $f_i(x,t)$ is the pdf of $T_i.a(t)$. If $F_i(x)$ is a differentiable, $f_i(x)$ is the derivative of $F_i(x)$.

Evaluation of p_i. We can now explain how p_i, the probability that $T_i.a$ is closest to q, is computed. In terms of X_i's the question is formulated as how p_i, the probability that X_i has the minimum value among all X_i's, is computed.

Let $\hat{f}_i(x)\,dx$ for indefinitely small dx be the probability that (1) $X_i \in [x, x+dx]$ and (2) $X_i = min_{1\leq k\leq |S|}X_k$. Then Equation 5.1 outlines the structure of our solution:

$$p_i = \int_{n_i}^{f} \hat{f}_i(x)\,dx \tag{5.1}$$

The $\hat{f}_i(x)\,dx$ is equal to probability $\mathrm{P}(X_i \in [x, x+dx])$ times the probability that X_i is the minimum among the all $X_i's$. The former probability is equal to $f_i(x)\,dx$ since $f_i(x)$ is X_i's pdf. The latter probability is equal to the probability that each X_j in S except for X_i have values greater than X_i, which equals to $\prod_{k=1\wedge k\neq i}^{|S|}\mathrm{P}(X_k > x)$, which also can be written as $\prod_{k=1\wedge k\neq i}^{|S|}\Big(1-F_k(x)\Big)$. Thus the formula for p_i can be written as:

$$p_i = \int_{n_i}^{f} f_i(x) \cdot \prod_{k=1\wedge k\neq i}^{|S|}\Big(1-F_k(x)\Big)\,dx \tag{5.2}$$

Observe that each $1-F_k(x)$ term registers the probability that $T_k.a$ is farther from q than $T_i.a$.

Efficient Computation of p_i The computation time for p_i can be improved. Note that $F_k(x)$ has a value of 0 if $x < n_k$. This means if $x < n_k$ then $1-F_k(x)$ is always 1, and T_k has no effect on the computation of p_i. Instead of always considering $|S|-1$ objects in the $\prod$ term of Equation 5.2 throughout $[n_i, f]$, we may actually consider fewer objects in some ranges, resulting in a better computation speed.

This can be achieved by first sorting the objects according to their *near_distance* from q. Next, the integration interval $[n_i, f]$ is broken down into a number of intervals, with end points defined by the *near_distance* of the objects. The probability of an object having a value of a closest to q is then evaluated for each interval in a way similar to Equation 5.2, except that we only consider $T_k.a$ with non-zero $F_k(x)$. Then

p_i is equal to the sum of the probability values for all these intervals. The final formula for p_i becomes:

$$p_i = \sum_{j=i}^{|S|} \int_{n_j}^{n_{j+1}} f_i(x) \cdot \prod_{k=1 \wedge k \neq i}^{j} \Big(1 - F_k(x)\Big)\, dx \tag{5.3}$$

Here we let $n_{|S|+1}$ be f for notational convenience. Instead of considering $|S| - 1$ objects in the $\prod$ term, Equation 5.3 only handles $j - 1$ objects in interval $[n_j, n_j + 1]$. This optimization is shown in Figure 5.6.

Example Let us use our previous example to illustrate how the evaluation phase works. After 4 objects $T_1, \ldots, T_4$ were captured (Figure 5.4(d)), Figure 5.8 shows the

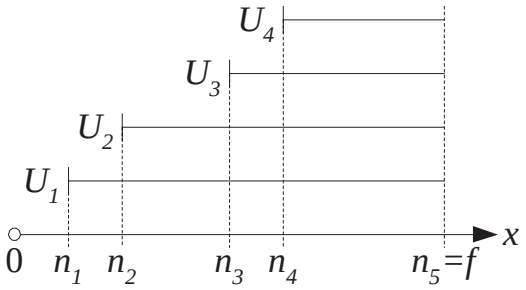


Figure 5.8 Illustrating the evaluation phase.

result after these objects have been sorted in ascending order of their *near_distance*, with the x-axis being the absolute difference of $T_i.a$ from q, and n_5 equals f. The probability p_i of each $T_i.a$ being the nearest neighbor of q is equal to the integral of $\hat{f}_i(x)$ over the interval $[n_i, n_5]$.

Let us see how we evaluate uncertainty intervals when computing p_2. Equation 5.3 tells us that p_2 is evaluated by integrating over $[n_2, n_5]$. Since objects are sorted according to n_i, we do not need to consider all 5 of them throughout $[n_2, n_5]$. Instead, we split $[n_2, n_5]$ into 3 sub-intervals, namely $[n_2, n_3]$, $[n_3, n_4]$ and $[n_4, n_5]$, and consider possibly fewer uncertainty intervals in each sub-interval. For example, in $[n_2, n_3]$, only U_1 and U_2 need to be considered.

5.3.3 Evaluation of EMinQ and EMaxQ

We can treat EMinQ and EMaxQ as special cases of ENNQ. In fact, answering an EMinQ is equivalent to answering an ENNQ with q equal to the minimum lower bound of all $U_i(T)$ in T. We can therefore modify the ENNQ algorithm to solve an EMinQ as follows: after the projection phase, we evaluate the minimum value of $l_i(t)$ among all uncertainty intervals. Then we set q to that value. We then obtain the results to the EMinQ after we execute the rest of the ENNQ algorithm. Solving an EMaxQ is symmetric to solving an EMinQ in which we set q to the maximum of $u_i(t)$ after the projection phase of ENNQ.

5.4 Evaluating value-based queries

In this section, we discuss how to answer the probabilistic value-based queries defined in Section 5.2.2.

5.4.1 Evaluation of VSingleQ

Evaluating a VSingleQ is simple, since by the definition of VSingleQ, only one object, T_k, needs to be considered. Suppose VSingleQ is executed at time t. Then the answer returned is the uncertainty information of $T_k.a$ at time t, i.e., $l_k(t)$, $u_k(t)$ and its pdf $f_k(x,t)$.

5.4.2 Evaluation of VSumQ and VAvgQ

Let us first consider the case where we want to find the sum of two uncertainty intervals $[l_1(t), u_1(t)]$ and $[l_2(t), u_2(t)]$ for objects T_1 and T_2. Notice that the values in the answer that have non-zero probability values lie in the range $[l_1(t) + l_2(t), u_1(t) + u_2(t)]$. For any x inside this interval, $f_X(x)$ (the pdf of random variable $X = T_1.a + T_2.a$) is:

$$f_X(x) = \int_{max\{l_1(t), x-u_2(t)\}}^{min\{u_1(t), x-l_2(t)\}} f_1(y,t) f_2(x-y,t)\, dy \tag{5.4}$$

The lower (upper) bound of the integration interval are evaluated according to the possible minimum (maximum) value of $T_1.a$.

We can generalize this result for summing the uncertainty intervals of $|T|$ objects by picking two intervals, summing them up using the above formula, and using the resulting interval to add to another interval. The process is repeated until we finish adding all the intervals. The resulting interval should have the following form:

$$\Big[\sum_{i=1}^{|T|} l_i(t), \sum_{i=1}^{|T|} u_i(t)\Big]$$

VAvgQ is essentially the same as VSumQ except for a division by the number of objects over which the aggregation is applied.

5.4.3 Evaluation of VMinQ and VMaxQ

To answer a VMinQ, we need to find the bounds of uncertainty region $[l, u]$, and pdf $f_X(x)$ of r.v. $X = \min_{1\leq i\leq |T|} T_i.a(t)$. Like for EMinQ the lower bound l can be set as $\min_{1\leq i\leq |S|} l_i(t)$ and upper bound u as $\min_{1\leq i\leq |S|} u_i(t)$, because X cannot take values outside $[l, u]$ The steps of the algorithm are similar to first three phases of ENNQ (projection, pruning, bounding) when q is set to be equal to l. Each T_i such that $l_i(t) > u$ is removed from set S of the relevant tuples. Then all tuples in S are sorted in ascending order of l_i. For notational convenience we introduce an additional parameter $l_{|S|+1}$ and set it equal to u.

To compute $f_X(x)$ notice that probability $\mathrm{P}(X \in [x, x + dx])$ for some small dx can be computed as sum of probabilities for each $T_i.a(t)$ to be inside $[x, x + dx]$ times the probability that the other $T_i.a(t)$'s are in $(x + dx, +\infty)$. As we tend dx to zero we have:

$$f_X(x) = \sum_{i=1}^{|S|}\Big(f_i(x,t) \cdot \prod_{k=1\wedge k\neq i}^{|S|}\Big(1 - F_k(x,t)\Big)\Big),\ \forall x \in [l, u] \tag{5.5}$$

Since if $x \in [l_j, l_{j+1}]$ and $k > j$, then $F_k(x, t) = 0$, therefore terms $(1 - F_k(x, t))$ are equal to 1 for such x's and k's and need not be considered by the formula. The

simplified formula is thus:

$$f_X(x) = \sum_{i=1}^{|S|}\Bigg(f_i(x,t) \cdot \prod_{k=1 \wedge k \neq i}^{j}\Big(1 - F_k(x,t)\Big)\Bigg),\ \forall x \in [l_j, l_{j+1}], 1 \leq j \leq |S| \tag{5.6}$$

Also notice that if $x \in [l_j, l_{j+1}]$ and $k > j$, then $f_i(x,t) = 0$. Thus formula for $f_X(x)$ can be written as:

$$f_X(x) = \sum_{i=1}^{j}\Bigg(f_i(x,t) \cdot \prod_{k=1 \wedge k \neq i}^{j}\Big(1 - F_k(x,t)\Big)\Bigg),\ \forall x \in [l_j, l_{j+1}], 1 \leq j \leq |S| \tag{5.7}$$

VMaxQ is handled in an analogous fashion.

5.5 Quality of probabilistic results

In this section, we discuss several metrics for measuring the quality of the results returned by probabilistic queries. It is interesting to see that different metrics are suitable for different query classes.

5.5.1 Entity-based non-aggregate queries

For queries that belong to the entity-based non-aggregate query class, it suffices to define the quality metric for each (T_i, p_i) individually, independent of other tuples in the result. This is because whether an object satisfies the query or not is independent of the presence of other objects. We illustrate this point by explaining how the metric of ERQ is defined.

For an ERQ with query range $[l, u]$, the result is the best if we are sure either $T_i.a$ is completely inside or outside $[l, u]$. Uncertainty arises when we are less than 100% sure whether the value of $T_i.a$ is inside $[l, u]$. We are confident that $T_i.a$ is inside $[l, u]$ if a large part of $U_i(t)$ overlaps $[l, u]$ i.e., p_i is large. Likewise, we are also confident that $T_i.a$ is outside $[l, u]$ if only a very small portion of $U_i(t)$ overlaps $[l, u]$ i.e., p_i is small. The worst case happens when p_i is 0.5, where we cannot tell if $T_i.a$ satisfies the range query or not. Hence a reasonable metric for the quality of p_i is:

$$\frac{|p_i - 0.5|}{0.5} \tag{5.8}$$

In Equation 5.8, we measure the difference between p_i and 0.5. Its highest value, which equals 1, is obtained when p_i equals 0 or 1, and its lowest value, which equals 0, occurs when p_i equals 0.5. Hence the value of Equation 5.8 varies between 0 to 1, and a large value represents good quality. Let us now define the *score* of an ERQ:

$$\text{Score of an ERQ} = \frac{1}{|R|} \sum_{i \in R} \frac{|p_i - 0.5|}{0.5} \tag{5.9}$$

where R is the set of tuples $(T_i.a, p_i)$ returned by an ERQ. Essentially, Equation 5.9 evaluates the average over all tuples in R.

Notice that in defining the metric of ERQ, Equation 5.8 is defined for each T_i, disregarding other objects. In general, to define quality metrics for the entity-based non-aggregate query class, we can define the quality of each object individually. The overall score can then be obtained by averaging the quality value for each object.

5.5.2 Entity-based aggregate queries

Contrary to an entity-based non-aggregate query, we observe that for an entity-based aggregate query, whether an object appears in the result depends on the existence of other objects. For example, consider the following two sets of answers to an EMinQ: $\{(T_1.a, 0.6), (T_2.a, 0.4)\}$ and $\{(T_1.a, 0.6), (T_2.a, 0.3), (T_3.a, 0.1)\}$. How can we tell which answer is better? We identify two important components of quality for this class: entropy and interval width.

Entropy. Let r.v. X take values from set $\{x_1, \ldots, x_n\}$ with respective probabilities $p(x_1), \ldots, p(x_n)$ such that $\sum_{i=1}^{n} p(x_i) = 1$. The entropy of X is a measure $H(X)$:

$$H(X) = \sum_{i=1}^{n} p(x_i) \log_2 \frac{1}{p(x_i)} \tag{5.10}$$

If x_i's are treated as messages and $p(x_i)$'s as their probability to appear, then the entropy $H(X)$ measures the average number of bits required to encode X, or the amount of information carried in X [Sha49]. If $H(X)$ equals 0, there exists some i such that $p(x_i) = 1$, and we are certain that x_i is the message, and there is no

uncertainty associated with X. On the other hand, $H(X)$ attains the maximum value when all the messages are equally likely, in which case $H(X)$ equals $\log_2 n$.

Recall that the result to the queries we defined in this class is returned in a set R consisting of tuples (T_i, p_i). Let r.v. Y take value i with probability p_i if and only if $(T_i, p_i) \in R$. The property that $\sum_{i=1}^{n} p_i = 1$ holds. Then $H(Y)$ measures the uncertainty of the answer to these queries; the lower the value of $H(Y)$, the more certain is the answer.

Bounding Interval. Uncertainty of an answer also depends on another important factor: the bounding interval B. Recall that before evaluating one of these aggregate queries, we need to find B that dictates all possible values we have to consider. Then we consider all the portions of uncertainty intervals that lie within B. Note that the decision of which object satisfies the query is only made within this interval. Also notice that the width of B is determined by the width of the uncertainty intervals associated with objects; a large width of B is the result of large uncertainty intervals. Therefore, if B is small, it indicates that the uncertainty intervals of objects that participate in the final result of the query are also small. In the extreme case, when the uncertainty intervals of participant objects have zero width, then the width of B is zero too. The width of B therefore gives us a good indicator of how uncertain a query answer is.

An example at this point will make our discussions clear. Figure 5.9 shows four different scenarios of two uncertainty intervals, $U_1(t_0)$ and $U_2(t_0)$, after the bounding phase for an EMinQ. We can see that in (a), $U_1(t_0)$ is the same as $U_2(t0)$. If we assume a uniform distribution for both uncertainty intervals, both T_1 and T_2 will have equal probability of having the minimum value of a. In (b), it is obvious that T_2 has a much greater chance than T_1 to have the minimum value of a. Using Equation 5.10, we can observe that the answer in (b) enjoys a lower degree of uncertainty than (a). In (c) and (d), all the uncertainty intervals are halved of those in (a) and (b) respectively. Hence (d) still has a lower entropy value than (c). However, since the uncertainty intervals in (c) and (d) are reduced, their answers should be more certain than those

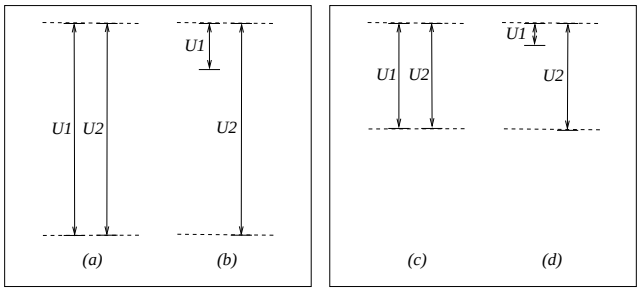


Figure 5.9 Illustrating how the entropy and the width of B affect the quality of answers for entity-based aggregate queries. The four figures show the uncertainty intervals ($U_1(t_0)$ and $U_2(t_0)$) inside B after the bounding phase. Within the same bounding interval, (b) has a lower entropy than (a), and (d) has a lower entropy than (c). However, both (c) and (d) have less uncertainty than (a) and (b) because of smaller bounding intervals.

of (a) and (b). Notice that the widths of B for (c) and (d) are all less than (a) and (b).

The quality of entity-based aggregate queries is thus decided by two factors: (1) entropy $H(Y)$ of the result set, and (2) width of B. We define their *scores* as follows:

$$\text{Score of an Entity, Aggr Query} = -H(Y) \cdot \text{width of } B \tag{5.11}$$

Notice that the query answer gets a high score if either $H(Y)$ is low, or the width of B is low. In particular, if either $H(Y)$ or the width of B is zero, then $H(Y) = 0$ is the maximum score.

5.5.3 Value-based queries

Recall that the results returned by value-based queries are all in the form an uncertainty interval $[l, u]$, and pdf $f(x)$. To measure the quality of such queries, we can use the concept of *entropy of a continuous distribution*, defined as follows:

$$\hat{H}(X) = -\int_l^u f(x) \log_2 f(x)\, dx \tag{5.12}$$

where $\hat{H}(X)$ is the entropy of continuous random variable X with pdf $f(x)$ defined in the interval $[l, u]$ [Sha49]. Similar to the notion of entropy, $\hat{H}(X)$ measures the

uncertainty associated with the value of X. Moreover, X attains the maximum value, $log_2(u-l)$ when X is uniformly distributed in $[l,u]$. Entropy $\hat{H}(X)$ can be negative, e.g. for uniform r.v. $X \sim \mathrm{U}[0,\frac{1}{2}]$.

We use the notion of entropy of a continuous distribution to measure the quality of value-based queries. Specifically, we apply Equation 5.12 to $f(x)$ as a measure of how much uncertainty is inherent to the answer of a value-based query. The lower the entropy value, the more certain is the answer, and hence the better quality is the answer. We now define the *score* of a probabilistic value-based query:

$$\text{Score of a Value-Based Query} = -\hat{H}(X) \tag{5.13}$$

The quality of a value-based query can thus be measured by the uncertainty associated with its result: the lower the uncertainty, the higher score can be obtained as indicated by Equation 5.13.

Please notice that though not presented here many different metrics are possible; e.g. one might choose the standard deviation as a metric for the quality of VMinQ, etc.

5.6 Improving answer quality

In this section, we discuss several update policies that can be used to improve the quality of probabilistic queries, defined in the last section. We assume that the sensors cooperate with the central server, i.e., a sensor can respond to update requests from the server by sending the latest value, as in the system model described in [OW02].

Suppose after the execution of a probabilistic query, some slack time is available before results are needed. The server can improve the quality of the answers to that query by requesting updates from sensors, so that the uncertainty intervals of some sensor data are reduced, potentially resulting in an improvement of the answer quality. Ideally, a system can demand updates from all sensors involved in the query; however, this is not practical in a limited-bandwidth environment. The question is

how to improve the quality with as few updates as possible. Depending on the types of queries, we propose a number of update policies.

Improving the Quality of ERQ The policy for choosing objects to update for an ERQ is very simple: choose the object with the minimum value computed in Formula 5.8, with an attempt to improve the score of ERQ.

Improving the Quality of Other Queries Several update policies are proposed for queries other than ERQ:

1. **Glb_RR.** This policy updates the database in a round-robin fashion using the available bandwidth, i.e., it updates the data items one by one, making sure that each item gets a fair chance of being refreshed.

2. **Loc_RR.** This policy is similar to Glb_RR, except that the round-robin policy is applied only to the data items that are related to the query, e.g., the set of objects with uncertainty intervals overlapping the bounding interval of an EMinQ.

3. **MinMin.** An object with its lower bound of the uncertainty interval equal to the lower bound of B is chosen for update. This attempts to reduce the width of B and improve the score.

4. **MaxUnc.** This heuristic simply chooses the uncertainty interval with the maximum width to update, with an attempt to reduce the overlapping of the uncertainty intervals.

5. **MinExpEntropy.** Another heuristic is to check, for each $T_i.a$ that overlaps B, the effect to the entropy if we choose to update the value of $T_i.a$. Suppose once $T_i.a$ is updated, its uncertainty interval will shrink to a single value. The new uncertainty is then a point in the uncertainty interval before the update. For each value in the uncertainty interval before the update, we evaluate the entropy, assuming that $U_i(t)$ shrinks to that value after the update. The mean

of these entropy values is then computed. The object that yields the minimum expected entropy is updated.

5.7 Experimental results

In this section, we experimentally study the relative behaviors of the various update policies described above with respect to improving the quality of the query results. We will discuss the simulation model followed by the results.

5.7.1 Simulation model

The evaluation is conducted using a discrete event simulation representing a server with a fixed network bandwidth ($\mathcal{B}$ messages per second) and 1000 sensors. Each update from a sensor updates the value and the uncertainty interval for the sensor stored at the server. The uncertainty model used in the experiments is as follows: An update from sensor T_i at time t_{update} specifies the current value of the sensor, $T_i.a_{srv}$, and the rate, $T_i.r_{srv}$ at which the uncertainty region (centered at $T_i.a_{srv}$) grows. Thus at any time instant, t, following the update, the uncertainty interval ($U_i(t)$) of sensor T_i is given by $T_i.a_{srv} \pm T_i.r_{srv} \times (t - T_i.t_{update})$. The distribution of values within this interval is assumed to be uniform.

The actual values of the sensors are modeled as random walks within the normalized domain as in [OW02]. The maximum rate of change of individual sensors are uniformly distributed between 0 and R_{max}. At any time instant, the value of a sensor lies within its current uncertainty interval specified by the last update sent to the server. An update from the sensor is necessitated when a sensor is close to the edge of its current uncertainty region. Additionally, in order to avoid excessively large levels of uncertainty, an update is sent if either the total size of the uncertainty region or the time since the last update exceed threshold values.

Note that the server has to *assume* certain model of change of values, e.g., it can assume the uniform or Gaussian distributions within each uncertainty interval. The actual change in values might follow a different distribution, and therefore the server

might make not very accurate predictions. We plan to address this problem in future work.

The representative experiments presented considered either EMinQ or VMinQ queries only. In each experiment the queries arrive at the server following a Poisson distribution with arrival rate λ_q. Each query is executed over a subset of the sensors. The subsets are selected randomly following the 80-20 hot-cold distribution (20% of the sensors are selected 80% of the time). The cardinality of each set was fixed at $N_{sub} = 100$. The maximum number of concurrent queries was limited to $N_q = 10$. Each query is allowed to request at most N_{msg} updates from sensors in order to improve the quality of its result.

In order to study different aspects of the policies, query termination can be specified either as (i) a fixed time interval (T_{active}) after which the query is completed even if its requested updates have not arrived (due to network congestion) or (ii) when a target quality ($\mathcal{G}$) is achieved. Depending upon the policy, we study either the average achieved quality (score), the average size of the uncertainty region, or the average response time needed to achieve the desired quality. All measurements were made after a suitable warm up period had elapsed. For fairness of comparison, in each experiment, the arrival of queries as well as the changes to the sensor values was identical.

Table 5.7.1 summarizes the major parameters and their default values. The simulation parameters were chosen such that average cardinality of the result sets achieved by the best update policies was between 3 and 10.

5.7.2 Results

All figures in this section show averages.

Bandwidth. Figure 5.10 shows scores for EMinQ achieved by various update policies for different values of bandwidth. The quality metric in this case is negated entropy times the size of the uncertainty region of the result set. Figure 5.11 is analogous

Table 5.2 Simulation parameters and their default values

Param	**Default**	**Meaning**
$\mathcal{D}$	$[0, 1]$	Domain of attribute a
R_{max}	0.1	Maximum rate of change of a ($\sec^{-1}$)
N_q	10	Maximum # of concurrent queries
λ_q	20	Query arrival rate (query/sec)
N_{sub}	100	Cardinality of query subset
T_{active}	5	Query active time (sec)
$\mathcal{B}$	350	Network bandwidth (msg/sec)
N_{msg}	5	Maximum # of updates per query
N_{conc}	1	The # of concurrent updates per query

to Figure 5.10 but shows scores for VMinQ instead of EMinQ. The score for VMinQ queries is negated continuous entropy.

In Figures 5.10 and 5.11, the scores increase as bandwidth increases for all policies, approaching the perfect score of zero for EMinQ. This is explained by the fact that with higher bandwidth the updates requested by the queries are received faster. Thus for higher bandwidth the uncertainty regions for freshly updated sensors tend to be smaller than those using lower bandwidth. Smaller uncertainty regions translate into smaller uncertainty of the result set, and consequently higher score. The reduction in uncertainty regions with increasing bandwidth can be observed from Figure 5.12.

All schemes that favor updates for sensors being queried significantly outperform the the only scheme that ignores this information: Glb_RR. The best performance is achieved by the MinMin policy, which updates a sensor with the lower bound of the uncertainty region $l_i(t)$ equal to the minimum lower bound among all sensors considered by the query. The MinExpEntropy policy showed worse results[1] than the

[1]The experiment with bandwidth of 200 takes too much time to complete.

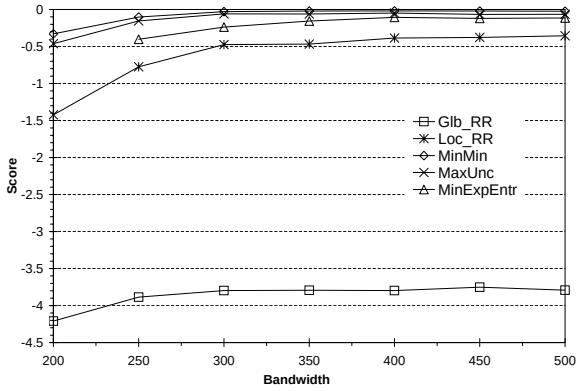


Figure 5.10 EMinQ score as function of $\mathcal{B}$

MinMin and MaxUnc policies in Figures 5.10 and 5.12 and worse results than those of the MinMin policy for VMinQ queries, Figure 5.11. When comparing the MinMin and MaxUnc policies, the better score of the MinMin policy is explained by the fact that the sensor picked for an update by the MinMin policy tends to have large uncertainty too – in fact, the uncertainty interval is at least as large as the width of the bounding interval. In addition the value of its attribute a tends to have higher probability of being minimum.

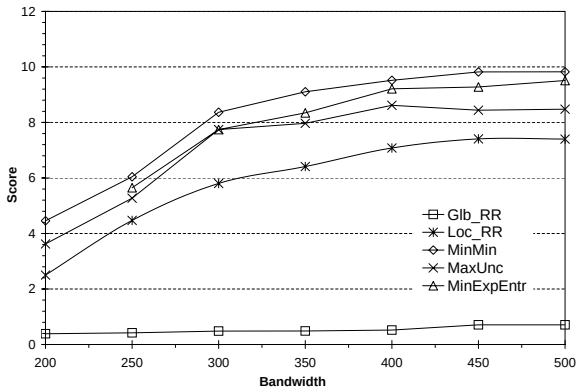


Figure 5.11 VMinQ score as function of $\mathcal{B}$

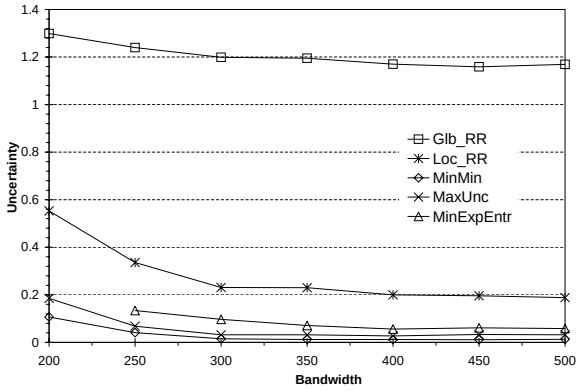


Figure 5.12 Uncertainty as function of $\mathcal{B}$

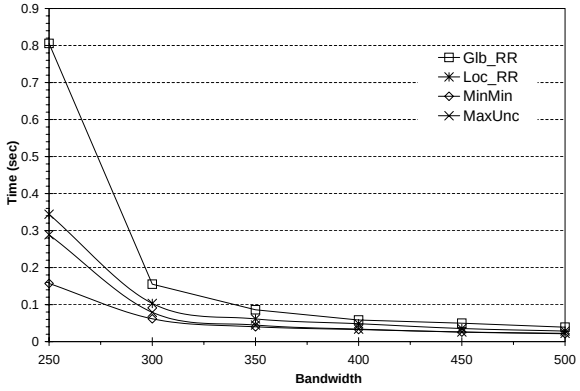


Figure 5.13 Response time as function of $\mathcal{B}$

Response Time. Figure 5.13 shows response time as a function of available bandwidth for EMinQ. Unlike the other experiments, in this experiment a query execution is stopped as soon as the goal score $\mathcal{G}$ (-0.06) is reached. Once again the MinMin strategy showed the best results, reaching the goal score faster than the other policies. The difference in response time is especially noticeable for smaller values of bandwidth, where it is almost twice as good as the other strategies. Predictably, the response time decreases when more bandwidth becomes available.

Arrival Rate. Figures 5.14 and 5.15 show the scores achieved by EMinQ and VMinQ queries for various update policies as a function of query arrival rate λ_q.

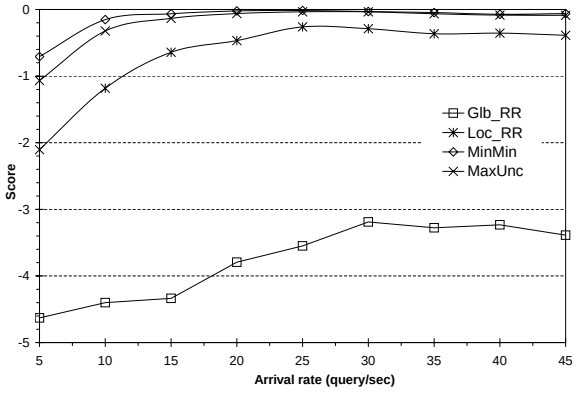


Figure 5.14 EMinQ score as function of λ_q

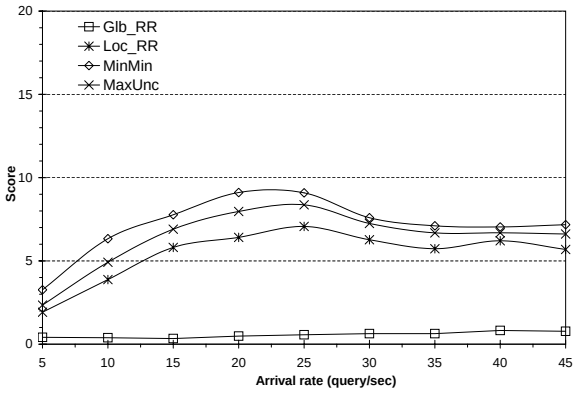


Figure 5.15 VMinQ score as function of λ_q

As λ_q increases from 5 to 25, more queries request updates and reduce the uncertainty regions. As a result, the uncertainty decreases, which leads to better scores (Figure 5.16). When λ_q reaches 25 the entire network bandwidth is utilized. As λ_q continue to increase queries are able to send fewer requests for updates and receive fewer updates in time, leading to poor result quality and larger uncertainty.

We can observe from Figures 5.14, 5.15, and 5.16 that the relative performance of the various policies remains the same over a wide range of arrival rates ($\lambda_q \in [5, 45]$).

The experiments show that all policies that favor query-based updates achieve much higher levels of quality. For the queries considered, the MinMin policy gives

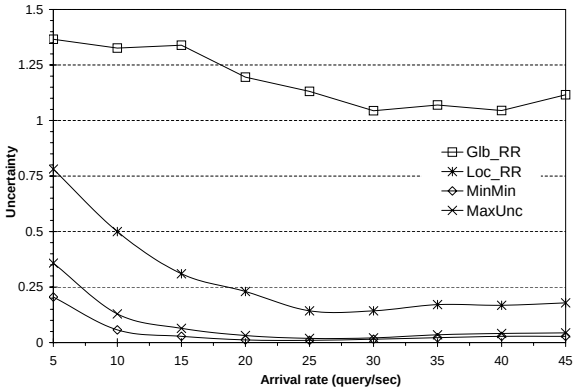


Figure 5.16 Uncertainty as function of λ_q

the best performance. Evaluation of the policies for all types of queries is beyond the scope of this thesis. We plan to address this issue as part of future work.

5.8 Related work

Many studies have focused on providing approximate answers to database queries. These techniques approximate query results based only upon a subset of data. In [VL94], Vrbsky et. al studied how to provide approximate answers to set-valued queries (where a query answer contains a set of objects) and single-valued queries (where a query answer contains a single value). An exact answer E can be approximated by two sets: a *certain set* C which is the subset of E, and a *possible set* P such that $C \cup P$ is a superset of E. Unlike our assumptions, their model assumes there is no uncertainty in the attribute values. Other techniques use precomputation [PG99], sampling [GM98] and synopses [AGPR99] to produce statistical results. While these efforts investigate approximate answers based upon a subset of the (exact) values of the data, our work addresses probabilistic answers based upon all the (imprecise) values of the data.

The problem of balancing the tradeoff between precision and performance for querying replicated data was studied by Olston et. al. [OW00, OLW01, OW02]. In their model, the cache in the server cannot keep track of the exact values of sensor

sources due to limited network bandwidth. Instead of storing the actual value for each data item in the server's cache, they propose to store an interval for each item within which the current value must be located. A query is then answered by using these intervals, together with the actual values fetched from the sources. In [OW00], the problem of minimizing the update cost within an error bound specified by aggregate queries is studied. In [OLW01], algorithms for tuning the intervals of the data items stored in the cache for best performance are proposed. In [OW02], the problem of minimizing the divergence between the server and the sources given a limited amount of bandwidth is discussed.

Khanna et. al [KT01] extend Olston's work by proposing an online algorithm that identifies a set of elements with minimum update cost so that a query can be answered within an error bound. Three models of precision are discussed: absolute, relative and rank. In the absolute (relative) precision model, an answer a is called α-precise if the actual value v deviates from a by not more than an additive (multiplicative) factor of α. The rank precision model is used to deal with selection problems which identifies an element of rank r: an answer a is called α-precise if the rank of a lies in the interval $[r - \alpha, r + \alpha]$.

In all the works that we have discussed, the use of probability distribution of values inside the uncertainty interval as a tool for quantifying uncertainty has not been considered. Discussions of queries on uncertainty data were often limited to the scope of aggregate functions. In contrast, our work adopts the notion of probability and provides a paradigm for answering general queries involving uncertainty. We also define the quality of probabilistic query results which, to the best of our knowledge, has not been addressed.

With the exception of [WSCY99], we are unaware of any work that discusses the evaluation of a query answer in probabilistic form. The study in [WSCY99] is limited to range queries for objects moving in straight lines in the context of a moving-object environment. We extend their ideas significantly by providing probabilistic

guarantees to general queries for a generic model of uncertainty. Other related work include [PXK+02, CPK03, KPHA02].

5.9 Conclusions

In this chapter we studied the problem of augmenting probability information to queries over uncertain data. We propose a flexible model of uncertainty, which is defined by (1) an lower and upper bound, and (2) a pdf of the values inside the bounds. We then explain, from the viewpoint of a probabilistic query, we can classify queries in two dimensions, based on whether they are aggregate/non-aggregate queries, and whether they are entity-based/value-based. Algorithms for computing typical queries in each query class are demonstrated. We present novel metrics for measuring quality of answers to these queries, and also discuss several update heuristics for improving the quality of results. The benefit of query-based updates was also shown experimentally.

6. CONCLUSION

This thesis addresses the problem of efficient querying of constantly evolving data. Constantly evolving data are common in moving object and sensor database environments.

Moving object environments are characterized by large numbers of moving objects and concurrent active queries over these objects. Efficient continuous evaluation of these queries in response to the movement of the objects is critical for supporting acceptable response times. We showed that neither the traditional approach, nor the brute force strategy achieve reasonable performance.

We presented a novel indexing technique for scalable execution: *Velocity Constrained Indexing (VCI)* and compared it with the *Query Indexing* technique. VCI is an indexing methods that help to avoid constant updates to the index as the data change. VCI approach gives good performance for up to a thousand of continuous queries, and, unlike the query indexing approach, it is unaffected by changes in queries and actual object movement. We showed that the two techniques complement each other enabling a combined solution that efficiently handles not only ongoing queries but also dynamically inserted queries. The experiments also demonstrated the robustness of the new techniques to variations in the parameters. The combined schemes therefore achieve superior performance to existing solutions for the efficient and scalable evaluation of continuous queries over moving objects.

We presented several approaches for in-memory evaluation of continuous range queries on moving objects. We established that the proposed method is in fact a very efficient solution even when there are no restrictions on object speed or nature of object movement or fraction of objects that move at any moment in time, which

are common restrictions made in similar research. We presented results for six different in-memory spatial indexes. The grid approach showed the best result for the uniform, skewed, and hyper-skewed cases. Even though the set of continuous queries is likely to remain almost unchanged, we also showed that nevertheless grid can very efficiently add and remove large numbers of queries. Overall, the Grid indexing is a highly scalable solution that gives orders of magnitude better performance than other index structures such as R*-trees.

We considered the problem of similarity join in main memory for low- and high-dimensional data. Two new algorithms were developed: *Grid-join* and *EGO*-join* that were shown to give superior performance than the state of the art EGO-join algorithm as well as RSJ. The significance of the choice of ε and recommendations for a good choice for testing and comparing algorithms with meaningful selectivity were discussed. While recent research has concentrated on joining high-dimensional data, little attention was been given to the choice of technique for low-dimensional data. In our experiments, the proposed Grid-join approach showed the best results for low-dimensional case or when values of ε are very small. The EGO*-join demonstrated substantial improvement over EGO-join for all the cases considered and is the best choice for high-dimensional data or when values of ε are large. The optimal choice of grid size was studied analytically and than corroborated experimentally.

Constantly evolving data is inherently uncertain. Querying such data with regular queries might lead to incorrect results. We investigated probabilistic methods of queries such data where each result has a probability of satisfying a query. We proposed a flexible model of uncertainty defined by uncertainty intervals and probability distribution of the values inside the intervals. We proposed a classification of queries based on the nature of query result set. Algorithms for computing typical queries in each query class are demonstrated. We showed that, unlike standard queries, probabilistic queries have the notion of quality of probabilistic answer. We present novel

metrics for measuring quality of answers to these queries, and also discuss several update policies for improving the quality of results. The benefit of query-based updates was shown experimentally.

To conclude, this thesis presented highly scalable and efficient state of the art solutions for the problems of handling multiple concurrent continuous range queries on moving objects and similarity joins. In addition to that it provided solutions to the problem of handling uncertainty in data for moving objects and sensor databases.

BIBLIOGRAPHY

[AAE00] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Dallas, Texas, May 2000.

[AAFZ95] Swarup Acharya, Rafael Alonso, Michael J. Franklin, and Stanley B. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 199–210, May 1995.

[AFZ96] S. Acharya, M. J. Franklin, and S. Zdonik. Disseminating updates on broadcast disks. In *Proceedings of the International Conference on Very Large Data Bases*, pages 354–365, September 1996.

[AGPR99] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 275–286, 1999.

[AHKP] W. Aref, S. Hambrusch, D. Kalashnikov, and S. Prabhakar. Pervasive location-aware computing environment. http://www.cs.purdue.edu/place.

[AHP00] W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Information management in a ubiquitous global positioning environment. Technical Report 00-006, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, February 2000.

[Ame94] N. Amenta. Bounded boxes, Hausdorff distance, and a new proof of an interesting Helly-type theorem. In *Proceedings of the Symposium on Computational Geometry*, pages 340–347, 1994.

[AS87] A. Aggarwal and S. Suri. Fast algorithms for computing the largest empty rectangle. In *Proceedings of the Symposium on Computational Geometry*, pages 278–290, 1987.

[AW88] A. Aggarwal and J. Wein. Computational geometry. Lecture Notes for MIT, 1988.

[BBBK00] C. Böhm, B. Braunmüller, M. Breunig, and H.-P. Kriegel. Fast clustering based on high-dimensional similarity joins. In *Proceedings of the International Conference on Information and Knowledge Management*, 2000.

[BBKK01] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–388, 2001.

[Ber98] P. Bernstein. The asilomar report on database research. *SIGMOD Record*, 27(4):74–80, 1998.

[BGO+96] 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.

[BK01] C. Böhm and H.-P. Kriegel. A cost model and index architecture for the similarity join. In *Proceedings of the International Conference on Data Engineering*, 2001.

[BKS93] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, May 1990.

[CGM90] Chris Clifton and Hector Garcia-Molina. Indexing in a hypertext database. In *Proceedings of the International Conference on Very Large Data Bases*, August 1990.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

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

[CPK03] Reynold Cheng, Sunil Prabhakar, and Dmitri V. Kalashnikov. Querying imprecise data in moving object environments. In *Proceedings of the IEEE International Conference on Data Engineering*, Bangalore, India, March 2003.

[FGNS00] L. Forlizzi, R. H. Guting, E. Nardelli, and M. Scheider. A data model and data structures for moving objects databases. In *Proceedings of the ACM SIGMOD Conference*, Dallas, Texas, May 2000.

[GBE+00] 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.

[GM98] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[GRS98] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[HJR97] Yun-Wu Huang, Ning Jing, and Elke A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *Proceedings of the International Conference on Very Large Data Bases*, August 1997.

[HLAP01] S. E. Hambrusch, C.-M. Liu, W. Aref, and S. Prabhakar. Minimizing broadcast costs under edge reductions in tree networks. In *Seventh International Symposium on Spatial and Temporal Databases (SSTD 2001)*, July 2001.

[HLL00] 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.

[HLL01] 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.

[IVB94] Tomasz Imieliński, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. In *Proceedings of the International Conference on Management of Data*, pages 25–36, May 1994.

[KCK01] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, May 2001.

[KGT99] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, June 1999.

[KH95] K. Koperski and J. Han. Discovery of spatial association rules in geographic information databases. In *International Symposium on Large Spatial Databases*, 1995.

[KH00] H. Koshima and J. Hoshen. Personal locator services emerge. *IEEE Spectrum*, 37(2):41–48, February 2000.

[KN98] E. M. Knorr and R. T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the International Conference on Very Large Data Bases*, 1998.

[KPAH02] 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, Indiana, June 2002.

[KPHA02] Dmitri V. Kalashnikov, Sunil Prabhakar, Susanne Hambrusch, and Walid Aref. Efficient evaluation of continuous range queries on moving objects. In *Proceedings of the International Conference on Database and Expert Systems Applications*, Aix en Provence, France, September 2002.

[KS98] Nick Koudas and Kenneth C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 466–475. IEEE Computer Society, 1998.

[KT01] S. Khanna and W. C. Tan. On computing functions with uncertainty. In *ACM Symposium on Principles of Database Systems*, 2001.

[KTF98] Anil Kumar, Vassilis J. Tsotras, and Christos Faloutsos. Designing access methods for bitemporal databases. *IEEE TKDE*, 10(1):1–20, 1998.

[LR96] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 247–258, June 1996.

[Ltd99] Trimble Navigation Ltd. Trimble customer solutions. http://www.trimble.com/solution/index.htm, 1999.

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

[MOS85] M. McKenna, J. O'Rourke, and S. Suri. Finding the largest rectangle in an orthogonal polygon. In *Proceedings of the Allerton Conference on Communication, Control, and Computing*, pages 486–495, 1985.

[OLW01] C. Olston, Boon Thau Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.

[OW00] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the International Conference on Very Large Data Bases*, 2000.

[OW02] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 73–84, 2002.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[PG99] V. Poosala and V. Ganti. Fast approximate query answering using precomputed statistics. In *Proceedings of the International Conference on Data Engineering*, page 252, 1999.

[PJ] D. Pfoser and C. S. Jensen. Querying the trajectories of on-line mobile objects. In *Proceedings of the MobiDE Conference, 2001.*

[PJ99] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-objects representations. In *Proceedings of the SSDBM Conference*, pages 123–132, 1999.

[PJT00] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[PTJ99] D. Pfoser, Y. Theodoridis, and C. S. Jensen. Indexing trajectories of moving point objects. Technical Report Chorochronos Technical Report CH-99-3, June 1999.

[PXK+02] 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.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, California, 1995.

[RR00] Jun Rao and Kenneth A. Ross. Making B^+-trees cache conscious in main memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 475–486, May 2000.

[SA97] John C. Shafer and Rakesh Agrawal. Parallel algorithms for high-dimensional similarity joins for data mining applications. In *Proceedings of the International Conference on Very Large Data Bases*, pages 176–185, August 1997.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.

[SDK01] Ayse Y. Seydim, Margaret H. Dunham, and Vijay Kumar. Location dependent query processing. In *Second ACM International Workshop on Data Engineering for Mobile and Wireless Access*, Santa Barbara, California, May 2001.

[Sha49] C. E. Shannon. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[SJLL00] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the position of continuously moving objects. In *Proceedings of the ACM SIGMOD Conference*, Dallas, Texas, May 2000.

[SSA97] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. High-dimensional similarity joins. In *Proceedings of the International Conference on Data Engineering*, pages 301–311, April 1997.

[SWCD97] 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.

[SWCD98] P. A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Querying the uncertain position of moving objects. In *Temporal Databases: Research and Practice*, number 1399. 1998.

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

[TUW98] Jamel Tayeb, Özgür Ulusoy, and Ouri Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[VL94] S. V. Vrbsky and J. W. S. Liu. Producing approximate answers to set- and single-valued queries. *The Journal of Systems and Software*, 27(3), 1994.

[WCD+98] 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, Florida, February 1998.

[WL98] Jay Werb and Colin Lanzl. Designing a positioning system for finding things and people indoors. *IEEE Spectrum*, 35(9):71–78, September 1998.

[Wol00] Ouri Wolfson. Research issues on moving object databases (tutorial). In *Proceedings of the ACM SIGMOD Conference*, page 581, Dallas, Texas, May 2000.

[WSCY99] 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.

[WXCJ98] Ouri Wolfson, Bo Xu, Sam Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proceedings of the SSDBM Conference*, pages 111–122, 1998.

[ZFAA94] Stanley Zdonik, Michael Franklin, Rafael Alonso, and Swarup Acharya. Are "disks in the air" just pie in the sky? In *IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.

[ZPBM98] J. M. Zagami, S. A. Parl, J. J. Bussgang, and K. D. Melillo. Providing universal location services using a wireless E911 location network. *IEEE Communications Magazine*, April 1998.