ABSTRACT

Title of Dissertation: Maintenance of Spatial Queries on Continuously Moving Points

Glenn S. Iwerks, Doctor of Philosophy, 2004

Dissertation directed by: Professor Hanan Samet
Department of Computer Science

Cars, aircraft, mobile cell phones, ships, tanks, and mobile robots all have the common property that they are moving objects. A kinematic representation can be used to describe the location of these objects as a function of time. For example, a moving point can be represented by the linear function $p(t) = \overrightarrow{x_0} + (t - t_0)\overrightarrow{v}$, where $\overrightarrow{x_0}$ is the start location, t_0 is the start time, and $\overrightarrow{v}$ is its velocity vector. Instead of storing the location of the object at a given time in a database, the coefficients of the function are stored. When an object's behavior changes enough that the function describing its location is no longer accurate, the function coefficients for the object are updated. Because the objects are represented as a function of time, spatial query results can change even when no transactions update the database. Our hypothesis is that algorithms for the maintenance of spatial queries on kinematic point data types can be developed to support updates to base relations as time advances that are more efficient than straight forward adaptations of previous work. We present algorithms to maintain k-nearest neighbor, spatial join, and spatial semijoin queries

in this domain. We compare by experimentation these new algorithms to more straight forward adaptations of previous work to support updates. Experiments are conducted using synthetic uniformly distributed data, and real aircraft flight data. The primary metric of comparison is the number of I/O disk accesses needed to maintain the query results and supporting data structures. A system to query and visualize results on moving object data, in a client-server environment, is also presented. The work presented here is built upon a culmination of our previously published work, including work on continuously moving point queries [35, 36], and client-server systems [31, 33, 34].

Maintenance of Spatial Queries

on Continuously Moving Points

by

Glenn S. Iwerks

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

Advisory Committee:

Professor Hanan Samet, Chairman/Advisor
Professor Samuel Goward
Professor David Mount
Professor V. S. Subrahmanian
Professor Amitabh Varshney

© Copyright by

Glenn S. Iwerks

2004

DEDICATION

To my loving wife Sydne, for being so patient.

TABLE OF CONTENTS

List of Tables vii

List of Figures ix

1 Introduction 1

2 Event Driven Query Processing 5

2.1 Data Types 6

2.2 Events 7

2.3 Notation 8

2.4 Event Example 9

2.5 Event Driven Query Processing Without Updates 9

3 Previous Work 15

3.1 Spreadsheet for Images 15

3.2 SAND Browser 16

3.3 The Original Spatial Spreadsheet 16

3.4 Spatial Queries on Static Data 18

3.5 Incremental View Maintenance 19

3.6 Incremental Distance Query 20

3.7 Moving Objects 21

3.7.1 Indexing Moving Objects 22

3.7.2 Moving Object Queries Over Time . . . 23
3.7.3 Time Parameterized Queries . . . 24
3.8 Animated Cartography . . . 27

4 The Internet Spatial Spreadsheet **29**
4.1 ISS Server . . . 29
4.2 ISS Client . . . 32
4.3 Example . . . 33
4.4 Cell Update Propagation . . . 35
4.5 Pushing Data From Server to Client Using HTTP . . . 42
4.6 Interactive Visualization . . . 45
4.7 Conclusion . . . 46

5 *K*-Nearest Neighbor Queries **48**
5.1 Continuous Windowing KNN (CW) . . . 49
5.2 Extending TP KNN for Updates . . . 59
5.3 Performance Issues . . . 64
5.4 Experimental Results . . . 68
5.5 Data Sets . . . 69
5.6 Experimental Results . . . 72
5.7 Conclusion . . . 79

6 Spatial Join Queries **80**
6.1 Query Engine . . . 83
6.2 All Events (AE) Approach . . . 88
6.3 Next Event (NE) Approach . . . 92
6.4 Performance Issues . . . 99
6.5 Experimental Results . . . 100

6.5.1 Implementation 100
6.5.2 Results 101
6.6 Conclusion 107

7 Spatial Semijoin Queries 110
7.1 Introduction 110
7.2 Data Structures 115
7.3 CFS Algorithm 118
7.4 CFS vs. CW 128
7.5 Experiments 129
7.6 Data Sets 130
7.7 Results 131
7.8 Concluding Remarks 136

8 Visualizing Changing Query Results for Moving Objects 138
8.1 Introduction 138
8.2 Definitions and Notation 140
8.3 Small Sets of Moving Objects 144
8.3.1 Fixed Update and Playback Rates 144
8.3.2 Example 146
8.3.3 Variable Update and Playback Rates 148
8.3.4 Variable Update Rate and Fixed Playback Rate 151
8.4 Large Sets of Intermittently Moving Objects 153
8.5 Large Sets of Continuously Moving Objects 156
8.6 Conclusion 157

9 Conclusion 159

A View Maintenance Proof **162**

A.1 Notation 162

A.2 Incremental Update of Spatial Join Views 164

A.2.1 Correctness 167

A.3 Rewrite Rule Proof 173

LIST OF TABLES

2.1 A trace of the Simple_Within() algorithm for the example in Figure 2.1 through time $t = 7$. 12

2.2 Simple_Nearest_Neighbor() algorithm trace for the example from Figure 2.1 through time $t = 7.5$. 14

4.1 Base relation r at time t_0. 34

4.2 Base relation s at time t_0. 34

4.3 Result of example query Q. The first row shows the initial result at time t_0. The second row shows the result at time $t_0 + 1$ after the deletion of tuple $\{y, (5,3), 1\}$, and insertion of tuple $\{y, (4,3), 1\}$ in relation s, and the deletion of tuple $\{a, (2,2), 1\}$ and insertion of tuple $\{a, (1,1), 1\}$ in relation r. The result at time $t_0 + 4$ is shown in the last row after tuple $\{y, (4,3), 1\}$ is replaced with tuple $\{y, (1,3), 1\}$ in s. Tuple $\{x, (3,1), 2\}$ from relation s does not appear in the join result because its *type* attribute is not equal to 1. 35

5.1 A trace of the CW algorithm applied to the example in Figure 5.2. . . . 53

5.2 ETP trace for the example in Figure 5.2. 61

5.3 Each column corresponds to a different aircraft data set. Each row is a statistic on the data sets. Row 1 is the mean number of flights at any given time (μ). Row 2 is the standard deviation in the number of flights (σ). Row 3 is the average update interval (UI) in seconds. 72

6.1 Trace of the example join query given in Figure 6.1. 82
6.2 Trace of the *All Events* (AE) algorithm for the example from Figure 6.1. 89
6.3 Trace of the *Next Event* (NE) algorithm for the example from Figure 6.1. 93
6.4 Each column corresponds to a different aircraft data set. Each row is a statistic on the data sets. Row 1 is the mean number of flights at any given time (μ). Row 2 is the standard deviation in the number of flights (σ). Row 3 is the average update interval (UI) in seconds. 100

7.1 Each column corresponds to a different aircraft data set. Each row is a statistic on the data sets. Row 1 is the mean number of flights at any given time (μ). Row 2 is the standard deviation in the number of flights (σ). Row 3 is the average update interval (UI) in seconds. 130

8.1 Example trace of procedure Convert 154

A.1 Truth table where T is TRUE and F is FALSE. The conditions $(\tau_s = \tau_t) =$ T and P$(\tau_s, \tau_v) \neq$ P(τ_t, τ_v) are not shown since these states are not possible. 176

LIST OF FIGURES

2.1 Example snapshots of 1D moving point attributes and events for time interval $1 \leq t \leq 7.5$. Arrow lengths indicate the distance traveled in one time unit. 10

2.2 Simple_Within() 12

2.3 Simple_Nearest_Neighbor() 13

3.1 The original Spatial Spreadsheet: Cells display spatial data contained in base relations and query results associated with each cell. 17

4.1 ISS data flow: $\mathbf{r}$ indicates a base relation, or materialized view. Tables i and d are insert and delete differential tables respectively. The query cell's view (a binary operation in this case) is initially computed using the base relations of each input cell. It is then incrementally updated when input cell's are updated. Changes in the query cell's view are stored in its own differential tables. This allows query cell's to be composed with other query cells. After the updates propagate through the system, the differential tables are applied to the cell's relation $\mathbf{r}$ and the differential tables are cleared for the next transaction. 31

4.2 Graphical representation of the state of relations r (locations denoted by the ■ symbol) and s (locations denoted by the • symbol) at times (a) t_0, (b) $\mathsf{t}_0 + 1$, and (c) $\mathsf{t}_0 + 4$. The ovals show pairs of objects included in the query result shown in Table 4.3. Arrows show the direction of motion of moving objects. 34

4.3 Update propagation example: The example query Q from Section 4.3 is broken up into two views. Supposing the user wants to see where all the objects of type 1 are located before the join is performed, a view is created as a selection on relation s using the query $\sigma s = \{\tau_s : \tau_s \in s \wedge \tau_s[type] = 1\}$ in cell (0,1). To see which objects are within distance 2 of each other, the user then creates the view $r \bowtie \sigma s = \{\tau_r \tau_{\sigma s} : \tau_r \in r \wedge \tau_{\sigma s} \in \sigma s \wedge \text{Distance}(\tau_r[loc], \tau_{\sigma s}[loc]) \leq 2\}$ in cell (1,1). Arrows indicate cell dependencies. 38

4.4 Update propagation data flow through cells in the ISS. The example shown is from Section 4.3, where the selection and join are separated out into different cells. The selection operation is $q_1 = \sigma s$, where σ is shorthand for $\sigma_{\tau_s[type]=1}$. The selection insert differential operation is $iq_1 = \sigma i_s$, and the selection delete differential operation is $dq_1 = \sigma d_s$. The join operation (a) is $q_2 = r \bowtie q_1$, where $\bowtie$ is shorthand for $\bowtie_{\text{Distance}(\tau_r[loc], \tau_s[loc]) \leq 2}$. The join insert differential operation (b) is $i_{q_2} = ((((i_r \bowtie q_1) \uplus (r \bowtie i_{q_1})) \uplus (i_r \bowtie i_{q_1})) - (i_r \bowtie d_{q_1})) - (i_{q_1} \bowtie d_r)$, and the join delete differential operation (c) is $d_{q_2} = (((q_1 \bowtie d_r) \uplus (r \bowtie d_{q_1})) - (d_{q_1} \bowtie d_r))$. 41

4.5 ISS client-server polling session. The client initiates the HTTP session. Updates generated in the server query threads are placed on a queue in the server. The server's push thread pops updates off the queue and transmits them to the client. The client push thread receives the updates and places them on the client's queue. The client data processing thread pops the data off the queue and executes the display callbacks. Each display callback is associated with a particular spreadsheet cell in the client. 44

5.1 2D example illustrating the CW approach, where $\otimes$ is the query point $\mathbf{q}$, r indicates the radius of the query window, $\bullet$ indicate points in $\mathbf{q}$'s within set, and $\circ$ indicate points not in the within set. 51
5.2 Example snapshots in time of 1D moving points, events, and updates up to time $t = 3.5$. Arrow length indicates distance traveled in one unit of time. The shaded area shows the extent of the query window within distance $d = 1.5$ of query point $\mathbf{q}$. 52
5.3 Continuous_Windowing_Knn() (CW) 52
5.4 CW_Adjust_Window() 54
5.5 CW_Compute_Knn_Result() 55
5.6 CW_Update_Data_Relation() 56
5.7 CW_Update_Query_Point() 57
5.8 CW_Process_Within_Evt() 58
5.9 CW_Process_Nn_Evt() 59
5.10 Extended_TP_Knn() (ETP) 61
5.11 ETP_Update_Data_Relation() 63
5.12 ETP_Update_Query_Point() 64
5.13 ETP_Process_Nn_Evt() 64

5.14 Example EB-tree with one root node, and two leaf nodes. 69
5.15 Snapshot of aircraft flight data. 71
5.16 Disk accesses vs. data set size for aircraft data. 74
5.17 Disk accesses vs. data set size for uniform data. 74
5.18 No updates 75
5.19 Number of entries vs. data set size for aircraft data. 76
5.20 Number of entries vs. data set size for uniform data. 76
5.21 Disk accesses vs. number of neighbors for aircraft data. 76
5.22 Disk accesses vs. number of neighbors for uniform data. 76
5.23 Disk accesses vs. extra neighbors for CW algorithm. 77
5.24 Disk accesses vs. number of disk cache pages for CW algorithm. 78
5.25 Disk accesses vs. number of disk cache pages for ETP algorithm. 78
5.26 Disk accesses vs. average update interval for uniform data. 79

6.1 Example spatial join of 1-dimensional moving points. 82
6.2 CSJU() 84
6.3 AE_Process_Next_Event(j) 90
6.4 AE_Insert_L(j) 91
6.5 AE_Delete_L() 91
6.6 AE_Generate_Events(j) 92
6.7 NE_Process_Next_Event(j) 94
6.8 NE_Insert_L(j) 95
6.9 NE_Insert_R(j) 97
6.10 NE_Delete_R() 98
6.11 NE_Generate_Events(j) 98

6.12 Comparison of total disk accesses for our simple adaptation of Tao and Papadias's CSJ algorithm to support updates (TP) to NE and AE without updates (a) and with updates (b and c). 104
6.13 Aircraft flight data (x-axis is log scale) 105
6.14 Uniform synthetic data 106
6.15 Number of disk accesses vs. join distance 107
6.16 Number of disk accesses vs. mean number of moving points (per relation) 108

7.1 Example fuzzy set, where $\otimes$ is the query point $\mathbf{q}$, $\bullet$ indicate points in $\mathbf{q}$'s fuzzy set, and $\circ$ indicate points not in the fuzzy set. 114
7.2 Example NN-B-tree with one root node, and two leaf nodes, where id_i = id(query_pt(e_i)). 116
7.3 E-queue_Insert() 117
7.4 E-queue_Delete_QueryPt() 117
7.5 E-queue_Delete_All_DataPt() 117
7.6 Process_Event() 119
7.7 Enqueue_Event() 120
7.8 Handle_Underflow() 121
7.9 Update_Fuzzy_Set() 123
7.10 Insert_Data_Point() 124
7.11 Delete_Data_Point() 126
7.12 Insert_Query_Point() 127
7.13 Delete_Query_Point() 127
7.14 Disk accesses with respect to data set size. 133
7.15 Disk accesses with respect to *k* with *circle_factor* = 2. 133
7.16 CFS algorithm parameters. (a) Disk accesses vs. *circle_factor*. (b) Disk accesses vs. fuzzy-set-interval. 134

8.1 Function Process_And_Sequence() appends a new animation layer frame as a query is processed. 146

8.2 Function Play_Sequence() renders an animation layer sequence. 146

8.3 Function Process_Variable_Rate() saves a transaction time for each layer frame. 149

8.4 Procedure Play_Variable_Rate() 150

8.5 Procedure Convert() converts a variable rate layer frame sequence to a fixed rate bitmap animation. 153

8.6 Function Process_Incremental() copies the previous layer frame and applies the changes from differential tables i_v, and d_v. 156

8.7 Function Play_Kinematic() 158

Chapter 1

Introduction

Cars, aircraft, mobile cell phones, ships, tanks, and mobile robots all have the common property that they are moving objects. Some example sources of moving object data are mobile networks, sensor networks, and remote sensors such as radar. A moving object database stores, indexes and queries moving object data. The challenge in moving object databases is in the query processing and visualization of data that is frequently updated.

Consider the following queries on moving point objects. For a cell phone, keep track of the nearest cell tower. For a suspect getaway car, keep track of the nearest police cruiser. For each airplane, keep track of every other airplane that is too close for safety. For each tank, keep track of each target that is within firing range. For each moving firetruck, keep track of the nearest mobile police unit. For each unmanned air vehicle, keep track of the nearest observation objective. These are all examples of queries that need to be maintained over time.

A view is a query maintained over time. Typically, when transactions update data in the base relations, updates are propagated to query results using incremental view maintenance algorithms [9, 25]. These algorithms operate under the assumption know as the *heuristic of inertia* [25] which states that updates to relations involve

only a small fraction of tuples found in a relation.

In the domain of continuously moving objects, the heuristic of inertia may not hold. Consider two representations for moving objects: *samples* and *kinematic* representations. The sampling representation involves taking a sample of the location of a moving object at periodic intervals. The time interval may or may not be the same between each sample. A series of samples for the same object is called a track. Each sample in a track is associated with a unique object id.

The heuristic of inertia only holds for a sample representation when objects move *intermittently*. Intermittently moving objects move occasionally and then stop for relatively long periods of time. Object data is updated periodically in the database when its location changes. Cars in a parking lot are examples of intermittently moving objects.

A kinematic[1] representation describes the location of an object as a function of time. For example, a moving point can be represented by the linear function $p(t) = \overrightarrow{x_0} + (t - t_0)\overrightarrow{v}$, where $\overrightarrow{x_0}$ is the start location, t_0 is the start time, and $\overrightarrow{v}$ is its velocity vector. Instead of storing the location of the object at a given time in a database, the coefficients of the function are stored. When an object's behavior changes enough that the function describing its location is no longer accurate, the function coefficients for the object are updated. This can help reduce the number of updates needed to keep track of an objects location as compared to the sampling approach. Kinematic representations have been used in this way to reduce network traffic load in the domain of distributed interactive simulation [11, 51]. Also known as *kinetic data structures* (KDS), they were pioneered in part by Basch, Guibas, and Hershberger [5] in their computational geometry publications on theoretical KDS algorithms.

[1]Kinematics is the branch of mechanics that studies the motion of a body or a system of bodies without giving any consideration to its mass or the forces acting on it.

When objects are moving continuously we assume a kinematic representation. Continuously moving objects are in constant motion, but we assume their velocity is updated only periodically so that the heuristic of inertia is applicable. For example, aircraft in flight moving between way-points. When their course and speed varies by some threshold amount, the database is updated to reflect the change.

Although the heuristic of inertia may be applicable with kinematic data types even when all the objects are in constant motion, they introduce another problem that must be addressed in the maintenance of query results. Because the objects are represented as a function of time, spatial query results can change even when no transactions update the database. Little research has been published addressing the issue of query maintenance of kinematic data types with updates to the base relations. Most research has focused on ad-hoc queries returning a query result for a particular moment in time, or on future queries that accumulate a query result of a time interval in the future (see Chapter 3 for more on previous work and references).

Our hypothesis is that algorithms for the maintenance of spatial queries on kinematic point data types can be developed to support updates to base relations as time advances that are more efficient than straight forward adaptations of previous work. We present algorithms to maintain k-nearest neighbor, spatial join, and spatial semi-join queries in this domain. We compare by experimentation these new algorithms to more straight forward adaptations of previous work to support updates. Experiments are conducted using synthetic uniformly distributed data, and real aircraft flight data. The primary metric of comparison is the number of I/O disk accesses needed to maintain the query results and supporting data structures.

A system to query and visualize results on moving object data, in a client-server environment, is also presented. The system architecture and visualization techniques give a context and motivation behind the primary focus of this thesis, that is, algorithms to maintain query results on continuously moving points with updates.

The work presented here is built upon a culmination of our previously published work, including work on continuously moving point queries [35, 36], and client-server systems [31, 33, 34].

The rest of this thesis is organized as follows. Chapter 2 gives some background on event-based query processing. Chapter 3 discusses previous work on relevant system, query processing, and visualization approaches. Chapter 4 gives an overview of our system architecture. Chapter 5 presents k-nearest neighbor maintenance algorithms and experiment results. Chapter 6 presents spatial join, and Chapter 7 presents semijoin algorithms and experiment results. Visualization of moving object queries is addressed in Chapter 8. Concluding remarks are given in Chapter 9.

Chapter 2

Event Driven Query Processing

In this chapter we present background in the basic principals of event driven query processing to maintain spatial query results on linear kinematic point objects as the points move. This chapter serves as a tutorial in the basic concepts needed to understand the query algorithms in other chapters. In this chapter we consider two types of queries. Maintenance of a *within* query maintains the set of all the points within a given distance of a query point as the point move. Maintenance of a *k-nearest neighbor* (k-nn) query maintains the set of the k closest points to a query point. Algorithms to support both these queries are presented in this chapter to illustrate the fundamental differences in processing of these two types of queries. Support for updates to the base relations are not addressed here. Instead, it is assumed the state of the database remains constant throughout the duration of the queries. Support for updates to the kinematic points in the database during query maintenance is addressed in subsequent chapters.

2.1 Data Types

A common representation for moving points are sampled locations (sometimes know as discretely moving points) [50]. For example, the motion of an aircraft can be represented by sampling its location using radar every 6 seconds and updating its position in the database each time the location of the aircraft changes. The problem with this representation is that the costs of updating the location of every aircraft in flight in a database every 6 seconds, and maintaining queries between updates, are prohibitive.

A *kinematic*[1] representation is an alternative to sampling. The movement of a kinematic point is represented as a function of time. In particular, the motion of a linear kinematic point is represented by the linear function $p(t) = \overrightarrow{x_0} + (t - t_0)\overrightarrow{v}$, where $\overrightarrow{x_0}$ is the start location, t_0 is the start time, and $\overrightarrow{v}$ its velocity vector. The coefficients of this function are stored in the database for each point. When the speed or direction of an object changes, the database is updated. For example, if an aircraft moving east at 500 miles per hour turns to head south, then the the function describing its motion is updated with a new velocity vector to reflect the new direction of travel. Errors that may arise due to discrepancies between the kinematic model of the objects motion, and the actual location of an object are beyond the scope of this thesis. Kinematic data types have been studied in other domains such as simulation (i.e, dead reckoning) to reduce network traffic in distributed simulations [11, 51], and computational geometry [5, 6, 24] also know as kinetic data structures, and spatial databases [49, 58, 65].

When objects are moving continuously we assume a kinematic representation. Continuously moving objects are in constant motion, but we assume their velocity

[1]Kinematics is the branch of mechanics that studies the motion of a body or a system of bodies without giving any consideration to its mass or the forces acting on it.

changes only occasionally. For example, aircraft in flight moving between way points.

2.2 Events

Events are used to maintain query results on kinematic points as time advances. Events are processed to keep the query result consistent as the points move. This is similar to event-driven simulation [22], but instead of maintaining a simulation state, events are used to maintain query results as time advances. To support the maintenance of the results of these queries we distinguish between two basic types of events: *within events* (w-event), and *order change events* (oc-events).

The first basic event type, the *within event* (w-event), occurs when two objects move to be at a given distance d to a query point. If a point is moving closer to the query point, then the w-event is called an *enter* event. If a point is moving farther away from the query point then the w-event is called an *exit* event. For a moving point, the time of a within event is based on solving the Euclidean distance equation $|p(time), q(time)| = d$ for *time*, where p and q are two moving points. This results in a closed form quadratic equation. See [53, 65] for more details on the computation of events between pairs of moving points.

The other basic event type of event is the *order change event* (oc-event). The oc-event occurs when two points change order with respect to their distance from a query point. For query point q, and two other points p_1 and p_2, the time of their oc-events is based on solving the equation $|p_1(time), q(time)| = |p_2(time), q(time)|$ for *time* (see [53, 65] for details). A special case of an oc-event is a *nearest neighbor event* (nn-event). Given a query point and its current k^{th} neighbor, the nn-event is the soonest oc-event to occur in the future out of all possible future oc-events between the query point, the kth neighbor, and any other given query point in the data set. For example, suppose that q is a query point, p_k is its current k^{th} neighbor,

and S is a set of moving points $S = \{s_1...s_n\}$. For each point $s_i \in S$, if s_i is closer to q than p_k, then the next oc-event e_i of point s_i occurs the next time when s_i moves to become farther from q than p_k. If s_i is farther from q than p_k, then the next oc-event e_i of point s_i occurs the next time when s_i moves to become closer to q than p_k. The next nn-event for q, p_k, and S is the soonest oc-event e_i of all next oc-events $\{e_1...e_n\}$. The time of the next nn-event is the next time in the future the kth neighbor of q will change given the set S.

2.3 Notation

A particular instance of a kinematic point is denoted $\mathsf{pt}(x_0, v, t_0)$, where x_0 is the start location, t_0 is the start time, and v_0 is the velocity vector. We assume an object-relational database environment in which a point kinematic data type is an attribute in a relation r, referred to as a *moving point*, or simply a *point*. For simplicity, and without loss of generality, we consider relations having one moving point attribute. The instance of a moving point attribute for some tuple $\tau \in r$ is denoted $\mathsf{P}(\tau)$. Each instance of a moving point attribute value has its own unique identifier. This allows us to index a relation on the point's id and retrieve the tuple to which the instance belongs. To indicate the tuple containing some point instance p we write $\mathsf{Tuple}(p)$. The Euclidean distance between point instances p and q at time t is $||p, q, t|| = |p(t), q(t)| = \sqrt{(q(t) - p(t))^2}$

A w-event instance is denoted as $\mathsf{w}(p, t)$ where p is the moving point, and t is the time of the event. It is important to remember that the query point and distance are part of a within query, and not explicitly represented in the w-event notation. An oc-event instance is denoted as $\mathsf{oc}(p, t)$ where p is the point involved, and t is the time. Likewise, the query point is part of the query and not explicitly represented in the oc-event notation. Additionally, it is important to note that the kth neighbor of

the query point is the other point involved in the oc-event. Since this is part of a k-nn query result, it also is not explicitly represented in the oc-event notation. For some event e (either a w-event or an oc-event), $\mathsf{P}(e)$ denotes the moving point explicitly represented in that event (e.g., $\mathsf{P}(\mathsf{w}(p,t)) = p$). The time of an event is denoted as $\mathsf{Time}(e)$ (e.g., $\mathsf{Time}(\mathsf{w}(p,t)) = t$). For a null event (denoted $e = \emptyset$), $\mathsf{Time}(\emptyset) = \infty$.

2.4 Event Example

Figure 2.1 shows snapshots of a 1-dimensional data set $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ of moving points, and a query point $\mathbf{q}$ at different instances of time where $\mathbf{a} = \mathsf{pt}(1, 1, 1)$, $\mathbf{b} = \mathsf{pt}(3.5, 1, 1)$, and $\mathbf{c} = \mathsf{pt}(6.5, -1, 1)$. Query point $\mathbf{q} = \mathsf{pt}(5.5, 0, 1)$, and query distance $d = 1.5$. The shaded area around $\mathbf{q}$ indicates the region along the line that is within distance $d = 1.5$ of $\mathbf{q}$. A w-event, $\mathsf{w}(\mathbf{b}, 2)$, takes place at time $t = 2$ when point $\mathbf{b}$ comes within distance $d = 1.5$ of query point $\mathbf{q}$. An oc-event $\mathsf{oc}(\mathbf{b}, 4)$ is shown at time $t = 4$ when point $\mathbf{b}$ is moving closer to $\mathbf{q}$ than point $\mathbf{c}$. At time $t = 4$, point $\mathbf{c}$ is the nearest neighbor prior to the event.

2.5 Event Driven Query Processing Without Updates

In this section we present some simple algorithms to maintain simple queries without any updates to the database over the duration of the query. This section serves as a tutorial to give the reader a better sense of how events are used in query processing, and of the properties of the different types of events.

Event-driven query processing is used to maintain queries on kinematic data

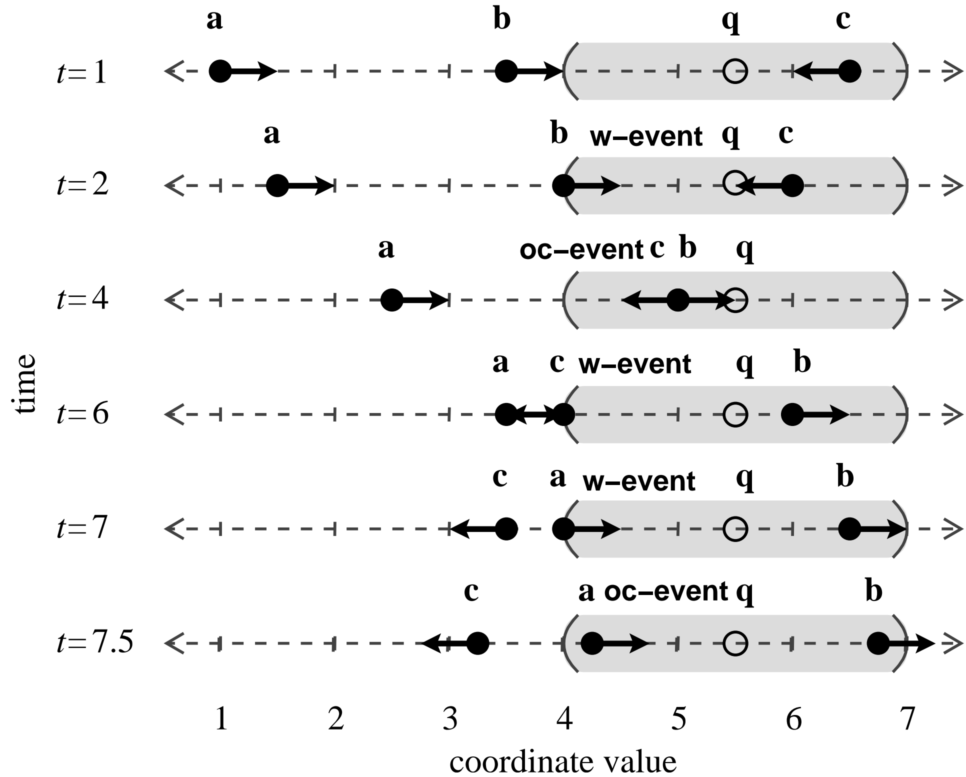


Figure 2.1: Example snapshots of 1D moving point attributes and events for time interval $1 \leq t \leq 7.5$. Arrow lengths indicate the distance traveled in one time unit.

types. This is similar to event-driven simulation [22], but instead of maintaining a simulation state, events are used to maintain query results as time advances. Events are processed in turn to keep query results consistent as points move.

Event-driven within query processing is performed by examining all within events in temporal order while updating the result appropriately[2]. Figure 2.2 gives a simple event-driven algorithm Simple_Within() for maintaining a within query. For example, consider the 1D scenario in Figure 2.1, and a query to find all points within distance $d = 1.5$ of q. Initially, relation r containing tuples with moving point attributes

[2]There are cases when a point may only "touch" the event threshold and then move back (closer or farther) to its former state. For simplicity, we do not address these cases.

$\{a, b, c\}$[3] is scanned (lines 2–6) to find the initial result at time $t = 1$, $W = \{c\}$, and the next w-event for each point. The w-events are inserted into priority queue Q, so that $Q = \{\mathsf{w}(b, 2), \mathsf{w}(c, 6), \mathsf{w}(a, 7)\}$. Function $\mathsf{next_w_event}(p, q, d, t)$ returns the next event after time t when point p will be at distance d from q, or it returns a null event with time stamp ∞ if no such event exists. This is a simple computation based on solving the equation $|p(time), q(time)| = d$ for *time*. For dimensionality greater than 1, this is a quadratic equation with a closed form solution. If the roots exist and are real numbers, then the next one greater than t is returned. Events are processed one-by-one (Figure 2.2, lines 7–14). Function $\mathsf{Pop}(Q)$ removes the next event from Q and returns it. Line 9–12 process enter events adding the incoming point to the within result and computing the next exit event. Line 13 processes exit events removing the point from the within result. No new events are generated by exit events. Table 2.1 shows a trace of the event processing portion of the algorithm (lines 7–14) up to time $t = 7$ for the 1D example in Figure 2.1 where $d = 1.5$.

An event-driven k-nn query processing algorithm finds the soonest oc-event to occur in the future out of all possible oc-events. This is called the *nearest neighbor event* (nn-event) because it will cause the k-nn query result to change. A nn-event is an oc-event, but not every oc-event is a nn-event. Figure 2.3 outlines a simple event-driven algorithm to maintain a simple nearest neighbor query.

The algorithm first scans r to find the nearest neighbor nn to query point q. The algorithm then examines every point to find the next oc-event for that point.

[3]For brevity only the moving attributes of tuples are shown.

```
     procedure Simple_Within(r, q, d, t)
1.     Q ← ∅,  W ← ∅
2.     foreach tuple τ ∈ r do
3.       if ||P(τ), q, t|| < d then W ← W ∪ τ
4.       e ←next_w_event(P(τ), q, d, t)
5.       if Time(e) < ∞ then Q ← Q ∪ e
6.     end foreach
7.     while Q ≠ ∅ do
8.       e ←Pop(Q),   t ←Time(e)
9.       if W∩Tuple(P(e)) = ∅ then
10.        W ← W∪Tuple(P(e))
11.        e ←next_w_event(P(e), q, d, t)
12.        if Time(e) < ∞ then Q ← Q ∪ e
13.      else W ← W−Tuple(P(e))
14.    end while
```

Figure 2.2: Simple_Within()

line #	e	Q	W	t
8	w$(b, 2)$	$\langle$w$(c, 6)$,w$(a, 7)\rangle$	$\{c\}$	2
10	”	”	$\{b, c\}$	”
11	w$(b, 8)$	”	”	”
12	”	$\langle$w$(c, 6)$,w$(a, 7)$,w$(b, 8)\rangle$	”	”
8	w$(c, 6)$	$\langle$w$(a, 7)$,w$(b, 8)\rangle$	”	6
13	”	”	$\{b\}$	”
8	w$(a, 7)$	$\langle$w$(b, 8)\rangle$	”	7
10	”	”	$\{a, b\}$	”
11	w$(a, 13)$	”	”	”
12	”	$\langle$w$(b, 8)$,w$(a, 13)\rangle$	”	”

Table 2.1: A trace of the Simple_Within() algorithm for the example in Figure 2.1 through time $t = 7$.

Function next_oc_event(p, q, nn, t) returns the next oc-event for point p after time t with respect to query point q, and the nearest neighbor nn. If no such event exists, then it returns a null event with time stamp ∞. Finding the time of oc-events is a simple computation based on solving the equation $|p(time), q(time)| =$

```
procedure Simple_Nearest_Neighbor(r, q, t)
1.   nn ← ∅, done ← false
2.   foreach tuple τ ∈ r do
3.     if nn = ∅ ∨ ||P(τ), q, t|| < ||P(nn), q, t|| then
4.       nn ← τ
5.   end foreach
6.   while ¬done do
7.     e ← ∅
8.     foreach τ ∈ r ∧ τ ≠ nn do
9.       e′ ←next_oc_event(P(τ), q,P(nn), t)
10.      if e = ∅∨Time(e′) <Time(e) then e ← e′
11.    end foreach
12.    if e ≠ ∅ then t ←Time(e), nn ←Tuple(P(e))
13.    else done ← true
14.  end while
```

Figure 2.3: Simple_Nearest_Neighbor()

$|nn(time), q(time)|$ for *time*. For dimensionality greater than 1 this is a quadratic equation with a closed form solution. If the roots exist and are real numbers, then the next one greater than t is returned. When the next nn-event comes due, the algorithm again examines every point and computes their oc-events to find the next nn-event. Table 2.2 shows a trace of the event processing portion of the algorithm (lines 6–14) up to time $t = 7.5$ for the 1D example in Figure 2.1.

Note that Simple_Nearest_Neighbor() does not have a queue for events. This is because the nearest neighbor changes on each nn-event thereby rendering previously computed oc-events irrelevant. The asymptotic running time for procedure Simple_Nearest_Neighbor() is $\mathcal{O}(E_{nn} * N)$ where E_{nn} is the number of nn-events processed throughout the course of the query maintenance, and N is the cardinality of r. The asymptotic running time for Simple_Within() is $\mathcal{O}(N + E_w)$ where E_w is the

line #	τ	e	e'	nn	t
7	-	$\emptyset$	-	c	1
8	a	"	"	"	"
9	"	"	oc$(a, 6.5)$	"	"
10	"	oc$(a, 6.5)$	"	"	"
8	b	"	"	"	"
9	"	"	oc$(b, 4)$	"	"
10	"	oc$(b, 4)$	"	"	"
12	"	"	"	b	4
7	"	$\emptyset$	"	"	"
8	a	"	"	"	"
9	"	"	oc$(a, 7.5)$	"	"
10	"	oc$(a, 7.5)$	"	"	"
8	c	"	"	"	"
9	"	"	oc(c, ∞)	"	"
12	"	"	"	a	7.5

Table 2.2: Simple_Nearest_Neighbor() algorithm trace for the example from Figure 2.1 through time $t = 7.5$.

number of w-events processed throughout the course of the query maintenance.

These simple algorithms serve to illustrate the fundamental differences in processing nn-events vs. processing w-events. The oc-events from which the nn-event is chosen are dependent on the query result which changes when an nn-event occurs. This makes all previous oc-events computed with respect to the old query result irrelevant. This requires computing new oc-events when the query result changes. On the other hand, pending w-events do not become irrelevant when the query result changes because w-events are independent of the query result.

Chapter 3

Previous Work

3.1 Spreadsheet for Images

Spreadsheets for Images (SI) [41] applies the spreadsheet concept to the image processing domain. In this case, the spreadsheet is a means of data visualization. Each cell in the spreadsheet contains graphical objects such as images and movies. Formulas for processing data can be assigned to cells. These formulas can use the contents of other cells as inputs. This ties the processing of data in the cells together. When a cell is modified, other cells that use it as input are updated. A similar capability is provided by the CANTATA programming language used with the KHOROS system [54].

3.2 SAND Browser

The SAND Browser is a front end graphical user interface for the SAND [19] spatial-relational database. The query results are displayed graphically. This gives the user an intuitive interface to the database to help the visualization of the data and the derivation of additional information from it. However, such a system does have limitations. In the SAND Browser, one primitive operation is processed at a time. A primitive operation is a query invoking one simple unary or binary query operation such as select, project, join, etc. When the user wants to make a new query, the results of the previous operation are lost unless saved explicitly in a new relation. As a result, there is no simple and implicit way to compose complex queries from primitives. In [31] we presented alternatives to the SAND Browser designed to overcome some of these limitations while maintaining ease of use and intuitive interface.

3.3 The Original Spatial Spreadsheet

In a classic spreadsheet, operations are single cell operations, row operations, or column operations. Column operations duplicate queries down a column of cells, and likewise for rows. The power of a spreadsheet is in its ability to organize data, formulate operations on that data quickly through the use of row and column operations, and to propagate changes in the data throughout the system. The original Spatial Spreadsheet described in [31] attempts to combine a spreadsheet paradigm with a spatial database management system, creating a new way to conceptually organize

spatial data, pose queries on that spatial data, and view the results (see Figure 3.1). In particular, the Spatial Spreadsheet provides a way to organize base relations and query results in a manner that is intuitively meaningful to the user.

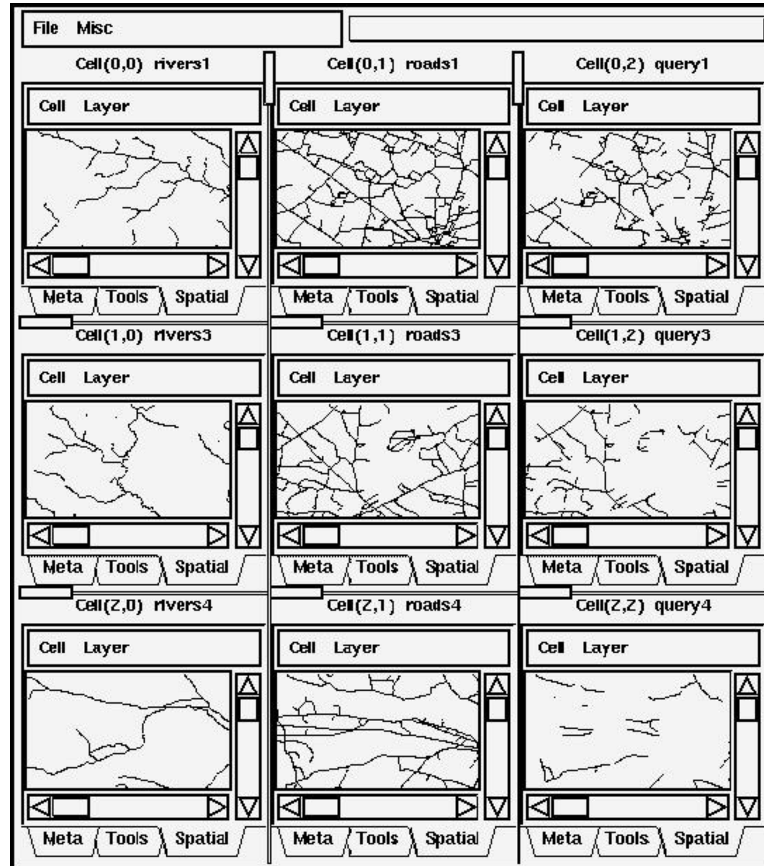


Figure 3.1: The original Spatial Spreadsheet: Cells display spatial data contained in base relations and query results associated with each cell.

The original Spatial Spreadsheet described in [31] operates as a front end to a spatial database system called SAND [19] running as a single process on one machine. In the classic spreadsheet paradigm, cell values are non-spatial data types whereas in the Spatial Spreadsheet, cell values are database relations. The Spatial Spreadsheet is made up of a 2D array of cells. Each cell in the Spatial Spreadsheet can be referenced by the cell's location (row, column). A cell can contain two types of relations: a base

relation, or a query result. A query result is a materialized view [9, 27] defined on the base relations or other materialized views. The user may pose simple queries using primitive query operators. For instance, a cell might contain the result of a spatial join between base relations in two other cells. Some examples of a primitive query operators are selection, projection, join, spatial join [29], window [4], nearest neighbor [30], etc. Primitive query operators may be composed to create complex queries. If a base relation is updated, the effects of those changes are propagated to all other cells by way of the query operators. This is done through the Spatial Spreadsheet's view maintenance algorithms. In the Spatial Spreadsheet, the spatial data are stored in a cell's relation can be displayed visually. This allows the effects of updates on the base relations to be observed when changes occur. Although the original Spatial Spreadsheet is useful as a means to organize, visualize, and query spatial data, it was not designed to handle dynamic spatial data, or operate as a remote client.

3.4 Spatial Queries on Static Data

Some of the most widely researched queries on static spatial data include within, window, spatial join, k-nearest neighbor (k-nn), and spatial semijoin. A *within* [59] query returns all objects within a given distance $d \geq 0$ from a query object. A *window* query can be thought of as a special case of a within query where the query object is a hyper-rectangle and the distance is zero. A *spatial join* [29] returns all pairs of objects in the Cartesian product of two relations that are within a given distance

$d \geq 0$ of each other. Each of these spatial queries also has an incremental [29, 30] version in which the query result is incrementally computed and reported one tuple at a time until some termination condition is satisfied. A k-nn query [56] returns the closest $k > 0$ spatial objects to a given query object. A *spatial semijoin* [29] is a subset of a spatial join $A \bowtie B$ where a tuple in the result $\langle a, * \rangle$ appears only once for any given $a \in A$, denoted $A \ltimes B$. An additional constraint is imposed in the spatial context of semijoins which stipulates for any tuple $\langle a, b \rangle$ in the result that $b \in B$ is the closest neighbor to a out of all objects in B. Another way to define this form of spatial semijoin is for every object $a \in A$, to find the nearest neighbor $b \in B$ and report $\langle a, b \rangle$. Using this definition, we can relax the 1-nn constraint and find the k nearest neighbors for every object $a \in A$, denoted $A \ltimes_k B$.

3.5 Incremental View Maintenance

There are several strategies for incremental view maintenance including *periodic* update, *deferred* update, and *immediate* update. Periodic updates are performed on *snapshots* [2, 42] only occasionally. This approach allows for fast queries and updates, but can only be used in systems that can tolerate stale data in the query results. Deferred updates [1, 15] delay updating views until the view is needed in a query. This allows faster updates but slower query processing times. Immediate updates are performed at the end of each transaction that modifies a base relation upon which a view is defined. Algorithms for immediate incremental view maintenance updates are presented in [9, 25]. A comparison of immediate, deferred and query modifica-

tion [63] is discussed in [27] along with a performance analysis by experiments. A system that uses immediate, deferred, and periodic update strategies in the same environment is described in [16].

3.6 Incremental Distance Query

The *incremental distance query* [30] returns all the objects within a given distance d of a query object q, one at a time, in increasing order of distance from q. The incremental distance query algorithm can be used for both within queries, and k-nn queries. Retrieving all the objects from q within distance $d < \infty$ is a within query. Retrieving the first k objects and then stopping, with $d = \infty$, is a k-nn query. The incremental distance query algorithm assumes a spatial tree index where, as in the case of the R-tree [26] spatial index, the internal nodes have bounding boxes (BB) that spatially contain all objects in the subtree. It makes use of a priority queue of objects sorted by distance from q. The queue is initialized with the root BB of the index. Objects are successively removed from the queue. Data objects are reported as they are dequeued. Internal nodes are expanded when they are dequeued by inserting each element in the node into the queue. This process continues until a maximum number of elements are reported, a maximum distance is reached, or there are no more elements in the queue.

3.7 Moving Objects

Objects represented as a function of time have been studied in other domains such as simulation [22, 51], and computational geometry [5]. In databases, past research includes indexing methods [3, 58, 66], ad-hoc queries [58, 62], and continuous queries such as continuous window [58, 62], within [35, 65], nearest neighbor and k-nearest neighbor (k-nn) [7, 35, 49, 53, 65], and spatial join [65].

The problem of querying objects (vehicles) moving through a road network is addressed in [14]. A space-time grid is used to partition and prune the domain space to support ad-hoc space-time range queries. In this work it is assumed that each object has a preplanned start point and destination. This means all future changes in speed and direction are computed and adjusted as objects are inserted. In the paper the spatial component of a grid cell is called a section. The max speed through a section is a function of the number of vehicles in the section. This can cause a cascade of updates through the grid if the insertion of an object changes the max speed in a section. Experimental results for algorithm performance is given. It is assumed all the data structures are main memory given that there is no mention of disk accesses, or secondary storage issues in the paper.

In [17] moving points are modeled as a function of time using the point-slope equation of a line. These lines are indexed in a space-time PMR quad-tree where time is one dimension and space is the rest. Ad-hoc space-time range queries are supported. No asymptotic analysis of the algorithms or experimental results are presented in the paper.

3.7.1 Indexing Moving Objects

In [3] three indexing schemes are presented to support ad-hoc window queries on continuously linearly moving 2D points. The first is a space-time partition tree, the second is a kinetisized range tree, and the third is a hybrid of the two. The kinetic range tree is more query efficient for range queries, but has more maintenance overhead. The third index is a trade off between the two. A modified version of the third index type can also answer approximate nearest neighbor queries. No experimental results are given, but extensive asymptotic analysis of each data structure is presented.

The *TPR-tree* [58] indexes moving objects described as a function of time. It is a disk-based object hierarchy R-tree variant. In the R-tree, each node is stored in one disk page. Each node has an associated minimum bounding box (MBB). Leaf nodes contain the MBBs for the indexed objects themselves. Each internal node has an MBB for each subtree spatially bounding the objects in the subtree. In the TPR-tree, a bounding box (BB) is a moving hyper-rectangle specified by two moving points defining opposite corners of the BB. The corner points are chosen so that the BB will always spatially contain the moving objects within it. The BBs in the TPR-tree rarely stay minimal, tending to grow faster than what would be the minimum bounding box at any given time. This is partly compensated for by the TPR-tree update algorithms. As an update occurs, the BB is adjusted to be minimal at the update time. Another compensatory action is that the TPR-tree insertion algorithm tries to insert objects moving in a similar manner (e.g., speed, direction), or to a

similar destination, into the same leaf node.

In [57] Šaltenis and Jensen present the R$^{\text{EXP}}$-tree. It is based on, and an improvement to the TPR-tree based on the assumption that most moving objects have a predetermined expiration time associated with them (x, v, t_{exp}). The idea in creation of bounding objects is to find a minimum bounding hyper-trapezoid, as it were, over the finite length trajectories of all the objects in a node. Expired entries are left in the tree until the node they are in is updated and written to disk by some other operation. The R$^{\text{EXP}}$-tree is compared experimentally with the TPR-tree in the paper.

3.7.2 Moving Object Queries Over Time

Plane-Sweep k-nn Algorithm

In [49], Mokhtar et. al. describe a method to maintain the k-nn query result on sets of moving point data over time. The algorithm starts by creating a list of points sorted by their current distance from a query point. Events are then computed corresponding to the instances in time when any point will change its position on the list with its neighbor. These events are inserted in a priority queue sorted by time. If an object is updated, then any events on the queue involving that object are recomputed. The list of points is updated as each event is processed in temporal order. The first k elements on the list form the k-nn result set.

The asymptotic size of the priority queue is $\mathcal{O}(n)$ where n is the number of moving points. The number of points that need to be examined when an event is processed

is $\mathcal{O}(1)$ since only immediate neighbors on the list need to be examined. If n is large, it would be reasonable to assume that much of the event queue would reside on disk. The rate of events that need to be processed in this approach depends on the distribution and motion characteristics of the data set, but it would be easy to imagine cases were the number of events processed over a given amount of time would be much greater than for the algorithm presented in Figure 2.3 because an event is processed every time any two objects change order on the list and not just when an object changes order with the nearest neighbor. The performance behavior in practice of this algorithm remains unclear since its presentation is theoretical, and no implementation details or experimental results are presented.

3.7.3 Time Parameterized Queries

Algorithms for processing moving object queries over a period of time into the future are presented in [65]. Known as *time-parameterized* (TP) queries, these queries take the current state of the database and predict how spatial queries will change in the future. TP queries do not support updates to the database. A TP query is of the form $\langle \mathbf{R}, \mathbf{T}, \mathbf{C} \rangle$ where $\mathbf{R}$ is the initial query result (e.g., at time 0), the *influence time* $\mathbf{T}$ is the time of the next event, and $\mathbf{C}$ is the set of objects involved in the event. The result $\mathbf{R}$ is the conventional component of the query. The event $\langle \mathbf{T}, \mathbf{C} \rangle$ is the called the TP component of the query.

Repeated application of a TP query yields a future query over a continuous period of time. The result is of the form $\langle \mathbf{R_1}, \langle \mathbf{T_1}, \mathbf{C_1} \rangle, \langle \mathbf{T_2}, \mathbf{C_2} \rangle, ..., \langle \mathbf{T_m}, \mathbf{C_m} \rangle \rangle$. This is

a future query giving the current result $\mathbf{R_1}$ followed by the future changes to the result given the current state of the database. An updated result $\mathbf{R_{i+1}}$ is derived by incrementally applying events $\langle \mathbf{T_i}, \mathbf{C_i} \rangle$ to the previous result $\mathbf{R_i}$. For example, suppose that an object o_1 is within query distance d of query object q at time t_0. Let the query result at time t_0 be R_{t_0}. Now suppose o_1 is moving away from q. At some time in the future $t_1 > t_0$, o_1 will be at exactly distance d from q. The event is $\langle t_1, o_1, q \rangle$. After time t_1 the query result R_{t_0} will be incorrect. To keep the query result correct, the result is updated, $R_{t_1} \leftarrow R_{t_0} - \{o_1\}$, at time t_1. Each subsequent event is determined through repeated application of the TP portion of the TP query. Each event involves one data object and the time it either enters the window, or leaves the window. The algorithm applies an incremental within event query to find each subsequent event.

An *incremental within event query* is similar to an incremental distance query [30], except that an event time metric is used instead of a distance metric [65]. An incremental within event query returns all the objects and the time at which they will enter the region within a given distance d around a query object q, one at a time, in increasing order of event time. If the distance $d = 0$ then the event time will be the time the objects will intersect, or cease to intersect one another. The algorithm assumes an object hierarchy tree index on the moving objects (e.g., the TPR-tree) for which internal nodes have bounding boxes (BB) that continually contain all the moving objects in each subtree. The algorithm is identical to the incremental distance query [30] (see above), except that the priority queue is sorted by within event time

instead of by the distance from q. The within event time for an internal node BB will always be less than or equal to the within event times of the objects it contains.

The *TP k*-nn (TP KNN) [65] is a k-nn query where $\mathbf{R}$ is the initial set of $k > 0, k \in \mathbb{N}$ closest objects to a given query object. The next event (TP portion of a query) is found using a next nearest neighbor event query.

A *next nearest neighbor event query* (next nn-event query) finds the next nearest neighbor given a query object and its current nearest neighbor. In [65], Tao and Papadias describe a method for finding the next nn-event given a query object, the current k^{th}neighbor, and a set of data points indexed in a TPR-tree. To find the next event, the bounding box (BB) of each node is examined and the node is placed on a global priority queue sorted by the oc-event time of its BB. Processing starts with the root node of the TPR-tree. The first object on the queue is dequeued and expanded, repeating the process recursively. When the first leaf node is examined, the object in the leaf with the soonest oc-event time is saved along with its event as the candidate nn-event. If the next BB on the queue has an oc-event sooner than the candidate nn-event, then it is expanded. And objects in a subsequent leaf node with an oc-event sooner than the candidate nn-event replaces the candidate. When the oc-event time of the next node on the queue is later than the candidate, then the candidate oc-event is returned as the next nn-event and processing stops.

The difference between TP KNN and TP WQ is that instead of the time being when objects become within a given distance from the query object, the event time is when an object is the same distance from the query object as the current k^{th} nearest

neighbor. This points out a fundamental difference between the within query and k-nn query when dealing with moving objects. In particular, events of TP KNN query are dependent on the query result, whereas, events of a TP WQ are not. This observation leads to a fundamentally different solution strategy when addressing the continuous k-nn query problem (see [65] for details).

Tao and Papadias's also describe a spatial join future query called the *continuous spatial join* (CSJ) algorithm in [65]. Each event involves two data objects and the time when they either start intersecting, or stop intersecting each other. Events are retrieved in increasing order of event time until some termination condition is satisfied (e.g., maximum time, maximum number of events, etc.).

3.8 Animated Cartography

Visualization of geo-referenced spatio-temporal data has been a topic of study for over 40 years [12]. One approach to this problem is to use static maps where temporal components are represented by different symbols or annotations on the map. Another approach is to use a chronological set of ordered maps to represent different states in time [40], sometimes known as strip maps. With the advent of more powerful computers and better graphics capabilities, animated maps are increasingly used to present time-series spatial data. One of the most commonly recognized uses is in presentation of weather reports on television. Animation is used in the presentation of meteorological data in weather forecast presentations to show changes over time [60]. Animated cartography is also used for decision support in disease

control to visually detect patterns and relationships in time-series geo-referenced health statistics [44, 45]. The use of animation in the study of remote sensing time-series data is also explored in [47, 48]. In [38], animated cartography is used in the presentation of urban environmental soundscape information for environmental decision support. The use of animation of spatio-temporal data in non-cartographic fields is presented in [37] and [43] to visualize dynamic scientific spatio-temporal data. The effectiveness of animation techniques to present time-series cartographic data to a user is studied in [39], where experiments were performed to determine the advantages of animated cartography over other presentation techniques. The study concluded that animation may be able to help decrease the amount of time needed for a user to comprehend time-series spatial data and to answer questions about it, over static map methods. In [20] a commercial system is described that supports 2D animation of base data, ad-hoc query processing, and triggers. However [20] does not describe any support for continuous or ongoing query processing or animation of continuous or ongoing query results, nor does it describe animations as interactive such that the perspective can be readily changed through zooming and panning while the animation is playing.

Chapter 4

The Internet Spatial Spreadsheet

The *Internet Spatial Spreadsheet* (ISS) extends the concept of the original Spatial Spreadsheet [31] to support dynamic spatial data, and operate over a network.

4.1 ISS Server

Conceptually, the ISS server manages a set of spreadsheets each consisting of a set of cells organized in rows and columns. Each cell manages the processing of a single relation. A cell's relation may be a base relation found in the database schema, or a materialized view. In the ISS, materialized view cells are also called *query cells*.

Incremental view maintenance techniques [23, 25, 55] have been extensively studied to efficiently maintain materialized views when changes occur. These techniques rely on the assumption that a relatively small number of tuples in an input relation are affected by any given transaction. This assumption is also known as the *heuristic of inertia* [25].

The data for each relation or materialized view is contained in three tables or relations. The first table is the main table containing the state of the entire base relation or query result. The other two tables contain pending updates to the main table. These are called the insert differential table, and the delete differential table. The main table is stored on disk. The differential tables are assumed to be small, so they are stored in main memory.

Consider a relation r. Let relation i_r be the set of tuples inserted into r during transaction Φ. The insertion update to r is expressed as $r' = r \uplus i_r$ where r' is the state of r after transaction Φ. Let relation d_r be the set of all tuples deleted from relation r during Φ. The deletion update to r is expressed as $r' = r - d_r$. By combining these two expressions we get $r' = (r \uplus i_r) - d_r$. The parentheses show the appropriate precedence needed in case a tuple is inserted and deleted during the same transaction. Symbols i_r, and d_r denote the insert differential table, and delete differential table of relation r respectively.

Incremental view maintenance algorithms are written by substituting $(r \uplus i_r) - d_r$ for r' in the query expression. For example, the update for the selection query $\sigma r'$ becomes $\sigma((r \uplus i_r) - d_r) = (\sigma r \uplus \sigma i_r) - \sigma d_r$. In this way the selection need only be applied to i_r (d_r), and the result inserted to (deleted from) the current query result σr. This results in an incremental update to the view rather than recomputing the view from scratch (see [25, 32] for more details).

Each cell of the ISS manages a main relation table, and two differential tables to support incremental view maintenance (see Figure 4.1). When a base relation is

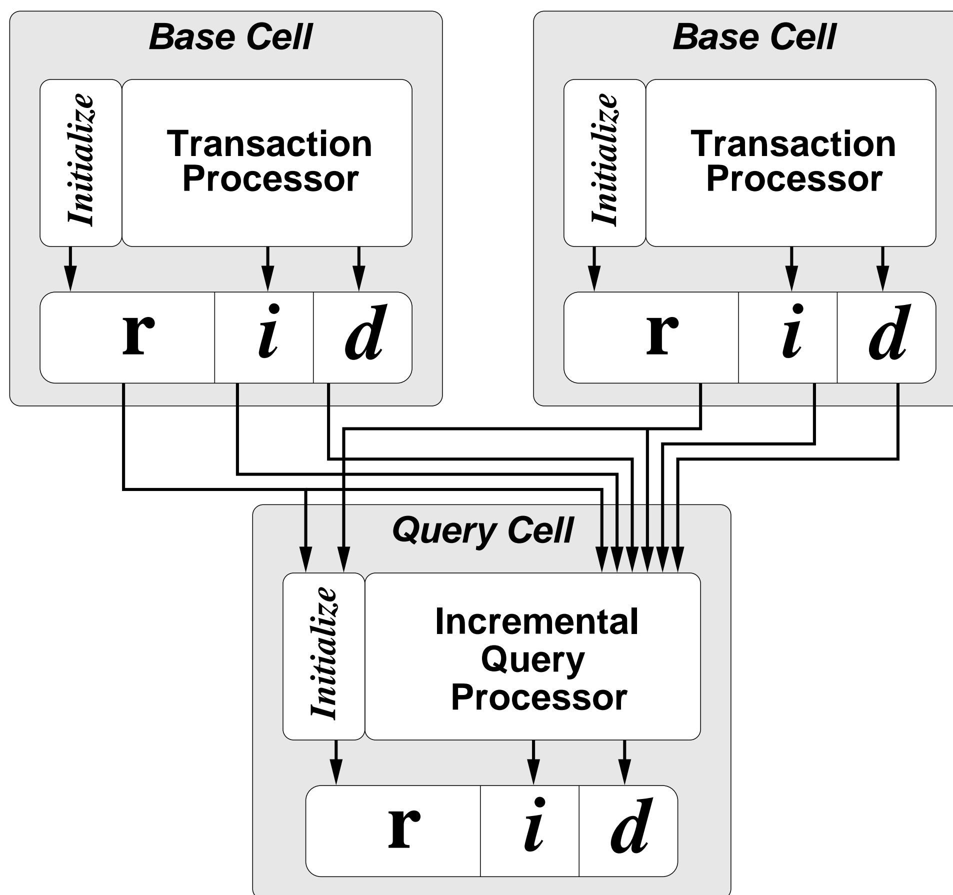


Figure 4.1: ISS data flow: **r** indicates a base relation, or materialized view. Tables i and d are insert and delete differential tables respectively. The query cell's view (a binary operation in this case) is initially computed using the base relations of each input cell. It is then incrementally updated when input cell's are updated. Changes in the query cell's view are stored in its own differential tables. This allows query cell's to be composed with other query cells. After the updates propagate through the system, the differential tables are applied to the cell's relation **r** and the differential tables are cleared for the next transaction.

opened in a cell, the cell handles the processing of updates to that base relation. An update is a combination of one or more insertions or deletions to a base relation that take place during a single database transaction. These updates then propagate to query cells managing the materialized views. The update propagation algorithm described in Section 4.4 is different from that of the original Spatial Spreadsheet presented in [31]. The original Spatial Spreadsheet did not update materialized

views incrementally, but instead recomputed the query results from scratch.

4.2 ISS Client

The ISS client runs remotely on a separate machine. The conceptual architecture of the client mirrors the server in that it also has a set of cells arranged in rows and columns. There is a one-to-one correspondence between the cells of a given client and the cells on the server. Cells in the client handle user interactions with the data. This includes query formulation, and spatial data visualization.

Instead of rendering an image on the server and transmitting it to the client, the server transmits the geometry information for the spatial data to the client, then the client renders the image on the client. Multiple perspectives, or different points of view, may be rendered simultaneously without increasing the load on the server. Each cell can have a different perspective through zooming and panning of individual cell displays. Additionally, when the perspective changes, the information needed to render the image is already on the client. If rendering were done on the server machine, a new image would need to be transmitted to the client each time the perspective changed. This would not only increase network traffic, but also increase the load on the server with work not directly related to query processing.

Although broadband Internet access is becoming more prevalent, improvements in processor speeds, main memory capacity, and graphics hardware are progressing even faster. This leads us to believe that network bandwidth rather than client machine processing and graphics capability is a greater constraint.

In our approach we allow for clients running behind firewalls. We assume only the ability to operate a client web browser that accesses external web sites from the client machine using the HTTP[21] protocol. We do not assume that any other means of communication through a firewall may be available. The HTTP client pull model presents some interesting challenges when there is a need to push data to the client from the server. This happens when base relations are updated thereby requiring data to be sent to the client to update the visualization displays.

4.3 Example

We will use the following query to illustrate the concepts presented in the following sections. Consider two relations $r(R)$ and $s(S)$ where schema $R = \{id, loc, type\}$, loc is a 2D point, id is a unique object identifier, and $type$ is a number. The schema of relation s is the same, $R = S$. As an example, consider the materialized view defined below. In our notation we denote a tuple in relation s as τ_s. For tuple $\tau_s \in s$ we denote the value of attribute α_0 in τ_s as $\tau_s[\alpha_0]$. The join of two tuples τ_q and τ_r is their concatenation is written $\tau_q \tau_r$.

$$Q = \{\tau_r \tau_s : \tau_r \in r \wedge \tau_s \in s \wedge \mathsf{Distance}(\tau_r[loc], \tau_s[loc]) \leq 2 \wedge \tau_s[type] = 1\}.$$

This query returns all pairs of objects in r and s that lie within 2 distance units of each other, where all the objects from s are of type 1. Suppose that the initial states of r and s at time t_0 are as shown in Tables 4.1 and 4.2, respectively. Now, suppose that object y is moving at a constant velocity, while object a moves and then stops as shown in the graphical representation in Figure 4.2. Intermittent updates to the

database change the current known locations of y and a.

id	*loc*	*type*
a	$(2,2)$	1
b	$(3.5,5)$	1
c	$(6,2)$	2

Table 4.1: Base relation r at time t_0.

id	*loc*	*type*
x	$(3,1)$	2
y	$(5,3)$	1
z	$(6,1)$	1

Table 4.2: Base relation s at time t_0.

Now consider the example query Q. The result for time t_0 is shown in the first row of Table 4.3. The locations of the objects participating in the join are indicated by the ovals in Figure 4.2a. Note that although object a is within distance 2 of object x, the pair is not included in the query result because the type of x is not 1.

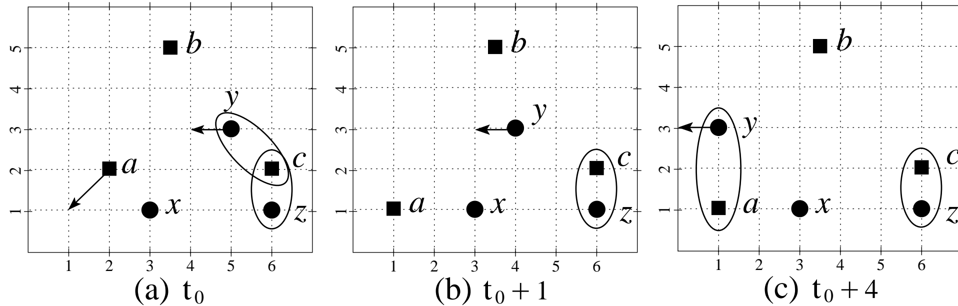


Figure 4.2: Graphical representation of the state of relations r (locations denoted by the ■ symbol) and s (locations denoted by the • symbol) at times (a) t_0, (b) t_0+1, and (c) t_0+4. The ovals show pairs of objects included in the query result shown in Table 4.3. Arrows show the direction of motion of moving objects.

Now suppose that at time $t_0 + 1$ minutes, the s relation is updated by deleting tuple $\{y,(5,3),1\}$ and inserting tuple $\{y,(4,3),1\}$, and the r relation is updated by deleting tuple $\{a,(2,2),1\}$ and inserting tuple $\{a,(1,1),1\}$. The resulting change in the query result is shown in the second row of Figure 4.3, and graphically by the oval in Figure 4.2b. Now suppose that after 3 more minutes, at time $t_0 + 4$, an update changes object y's location from $(4,3)$ to location $(1,3)$. The join result after the update is shown in the third row of Figure 4.3, and corresponding ovals in

Figure 4.2c.

at time:	$r.id$	$r.loc$	$s.id$	$s.loc$
t_0	c	$(6,2)$	y	$(5,3)$
	c	$(6,2)$	z	$(6,1)$
t_0+1	c	$(6,2)$	z	$(6,1)$
t_0+4	a	$(1,1)$	y	$(1,3)$
	c	$(6,2)$	z	$(6,1)$

Table 4.3: Result of example query Q. The first row shows the initial result at time t_0. The second row shows the result at time t_0+1 after the deletion of tuple $\{y,(5,3),1\}$, and insertion of tuple $\{y,(4,3),1\}$ in relation s, and the deletion of tuple $\{a,(2,2),1\}$ and insertion of tuple $\{a,(1,1),1\}$ in relation r. The result at time t_0+4 is shown in the last row after tuple $\{y,(4,3),1\}$ is replaced with tuple $\{y,(1,3),1\}$ in s. Tuple $\{x,(3,1),2\}$ from relation s does not appear in the join result because its *type* attribute is not equal to 1.

4.4 Cell Update Propagation

If any base relations used in the definition of the view are updated, then the view needs to be updated to reflect the change to keep query results current. To update materialized views after a transaction, it is often more efficient to reevaluate the query in terms of changes to the base relations instead of reevaluating the query from scratch.

This is known as incremental view maintenance [13, 23, 25, 55]. In particular we address the immediate update of materialized views as opposed to lazy, or deferred maintenance. A view is nested when it is in turn used in the definition of another view. We assume views may be nested up to a finite, non-cyclic, arbitrary depth. Tuple updates are treated as a deletion followed by an insertion of the updated

tuples. Duplicate tuples are allowed. In the case of duplicates, a deletion removes only one copy of a tuple. Additionally, a tuple may be inserted and deleted in the same transaction with the result of no net change.

When a transaction updates a base relation, the tuples to be inserted are entered into the insert differential table of the base relation before the transaction commits. Similarly, tuples to be deleted are entered into the base relation's delete differential table. When some view v is defined in terms of some view or base relation r, we say that v is a *dependent* of r. When the transaction commits, the contents of the differential tables for a given view, or base relation, are fed to its dependents. Updates for each subsequent materialized view are computed and inserted into its own differential tables. The contents of these are in turn propagated to other dependent views, and so on. The chain of dependents can not be cyclic. When the propagation is complete, the differential tables are applied to their associated main tables and the differential tables are cleared.

Cell Marking: To ensure correct order of execution, a simple cell marking algorithm is employed. Before a transaction takes place, all cells are marked *clean*. When a base relation is updated it is marked *dirty*. After all the base relations affected by the transaction are processed, and before the updates are propagated to their dependents, each dependent of a dirty base relation is marked dirty as well. Next, all of their dependent's dependents are marked dirty, etc. This process continues recursively until no more views can be marked.

To illustrate, consider the example query Q given in Section 4.3. To pose this

query in the ISS, the user first opens the base relations in their own cells. Figure 4.3 shows relation s open in cell (0,0), and relation r open in relation (1,0). Now suppose that the user wants to know where all the objects of type 1 are located before the join is performed. To do this the user can create a view using the query $\sigma s = \{\tau_s : \tau_s \in s \wedge \tau_s[type] = 1\}$. In Figure 4.3, σs is in cell (0,1). Finally, to see which objects are within distance 2 of each other, the user creates a view in cell (1,1) using the query $r \bowtie \sigma s = \{\tau_r \tau_{\sigma s} : \tau_r \in r \wedge \tau_{\sigma s} \in \sigma s \wedge \mathsf{Distance}(\tau_r[loc], \tau_{\sigma s}[loc]) \leq 2\}$. In this case the view $r \bowtie \sigma s$, shown in cell (1,1) of Figure 4.3, is equivalent to the example query Q given in Section 4.3.

Now suppose, once the spreadsheet is set up, a transaction arrives at time $\mathrm{t}_0 + 1$ minutes updating both relations r and s. When this occurs, the marking algorithm marks cell (0,0) and cell (1,0) as dirty. Once all base relations are marked, it then marks their dependents dirty, cells (0,1) and (1,1) in this case.

In the next step of update propagation, all cells managing base relations are marked clean. Then each query cell is examined. If all the cell's dependents are clean, then the incremental view maintenance for the cell's view is executed. No updates for a given materialized view may be computed until its dependents are marked clean. Views are marked clean as soon as the update for the view is calculated and stored in its differential tables. If instead of using this cell marking algorithm we simply iterate through the cells and update a view as soon as a dependent is updated, then incorrect results may occur. For example, without cell marking, cell (1,1) could be updated as a result of the update to cell (1,0) before cell (0,1) is updated. Since cell

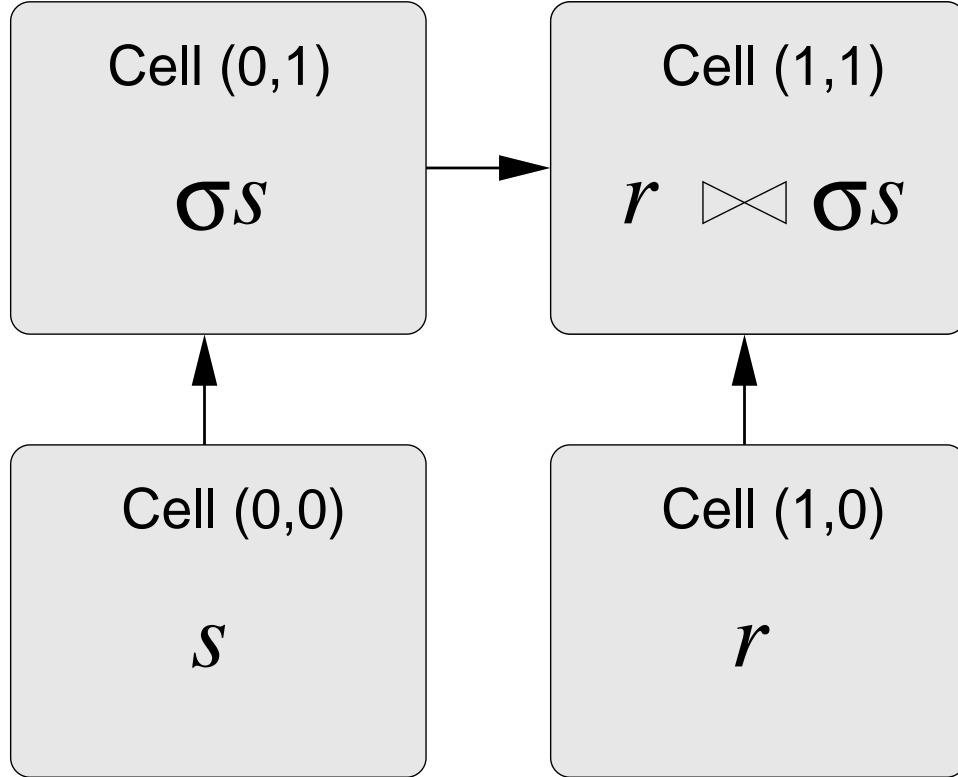


Figure 4.3: Update propagation example: The example query Q from Section 4.3 is broken up into two views. Supposing the user wants to see where all the objects of type 1 are located before the join is performed, a view is created as a selection on relation s using the query $\sigma s = \{\tau_s : \tau_s \in s \wedge \tau_s[type] = 1\}$ in cell (0,1). To see which objects are within distance 2 of each other, the user then creates the view $r \bowtie \sigma s = \{\tau_r \tau_{\sigma s} : \tau_r \in r \wedge \tau_{\sigma s} \in \sigma s \wedge \text{Distance}(\tau_r[loc], \tau_{\sigma s}[loc]) \leq 2\}$ in cell (1,1). Arrows indicate cell dependencies.

(1,1) is also dependent on cell (0,1) this could lead to incorrect results.

View Maintenance: Consider a relation r. Let relation i_r be the set of tuples inserted into r during transaction Φ. The insertion update to r is expressed as $r' = r \uplus i_r$. Let relation d_r be the set of all tuples deleted from relation r during Φ. The deletion update to r is expressed as $r' = r - d_r$. By combining these two expressions we get $r' = (r \uplus i_r) - d_r$. The parentheses show the appropriate precedence needed in case a tuple is inserted and deleted during the same transaction. Symbols i_r, and d_r denote to the insert differential table, and delete differential table in a cell

respectively.

Consider some arbitrary binary query operator Υ that operates on two relations. Suppose $v = l \Upsilon r$ is our original view definition. Let $v' = l' \Upsilon r$ denote the query update operation after relation l is updated by some transaction Φ where l' is the state of relation l after Φ is applied. By substitution, this expression can be rewritten in terms of l's differential tables and the state of l before Φ as $v' = ((l \uplus i_l) - d_l) \Upsilon r$. If the operator Υ is distributive over the $\uplus$ and $-$ operations, then this expression can be rewritten as $v' = ((l \Upsilon r) \uplus (i_l \Upsilon r)) - (d_l \Upsilon r)$. Substituting v for $(l \Upsilon r)$ the expression becomes $v' = (v \uplus (i_l \Upsilon r)) - (d_l \Upsilon r)$. The operations $(i_l \Upsilon r)$ and $(d_l \Upsilon r)$ are much easier to compute than $(l' \Upsilon r)$ since by the heuristic of inertia we can assume that i_l, and d_l are much smaller than l'. Therefor, i_l and d_l are stored in main memory making access to the table's contents much faster.

Instead of imediately applying the update to the main table, the changes are stored in auxiliary differential tables associated with v where $i_v = (i_l \Upsilon r)$ and $d_v = (d_l \Upsilon r)$. These differential tables are then fed into any number of nested views defined in terms of v. The approach is similar for unary query operators.

Computing differential tables for binary operators, when both of the relations change in the same transaction, is more difficult. One approach is to apply the updates to one base relation (say l), run the update propagation algorithm, then apply the updates to r, and run the update propagation algorithm again. This approach can result in executing the update propagation as many times as there are bases relations updated in a single transaction.

A different approach ensures that the update propagation algorithm is run only once regardless of the number of base relations that are updated. It is derived by factoring out the term $(l \Upsilon r)$ from the equation $v' = ((l \uplus i_l) - d_l) \Upsilon ((r \uplus i_r) - d_r)$ so v can be substituted for $(l \Upsilon r)$. The resulting expression (see Expression 4.4.1) is more complex than when only one table is updated, but still leads to an efficient view maintenance algorithm that can be nested. Expression 4.4.2 is the subexpression for the insert differential table of the binary view operation, and Expression 4.4.3 is the subexpression for the delete differential table. Note that only two terms in expression 4.4.2, and two terms in expression 4.4.3 involve base relations. None of these involve more than one base relation. All other terms are operations between differential tables stored in main memory.

A proof of correctness for Expression 4.4.1 is given in Appendix A.

Expression 4.4.1

$$\begin{aligned} v' = (v \uplus (((((i_l \Upsilon r) \uplus (l \Upsilon i_r)) \uplus (i_l \Upsilon i_r)) - (i_l \Upsilon d_r)) - (i_r \Upsilon d_l))) \\ - (((r \Upsilon d_l) \uplus (l \Upsilon d_r)) - (d_r \Upsilon d_l)) \end{aligned}$$

Expression 4.4.2

$$i_v = ((((i_l \Upsilon r) \uplus (l \Upsilon i_r)) \uplus (i_l \Upsilon i_r)) - (i_l \Upsilon d_r)) - (i_r \Upsilon d_l)$$

Expression 4.4.3

$$d_v = (((r \Upsilon d_l) \uplus (l \Upsilon d_r)) - (d_r \Upsilon d_l))$$

Figure 4.4 shows the update propogation algorithm for our join example using these expressions.

To evaluate either expression without indexes, in the worst case, the number of disk page accesses is $\mathcal{O}(n_r/N_r + n_l/N_l)$ where n_r (n_l) is the number of tuples in relation r (l), and N_r (N_l) is the number of tuples in r (l) that will fit in a single disk

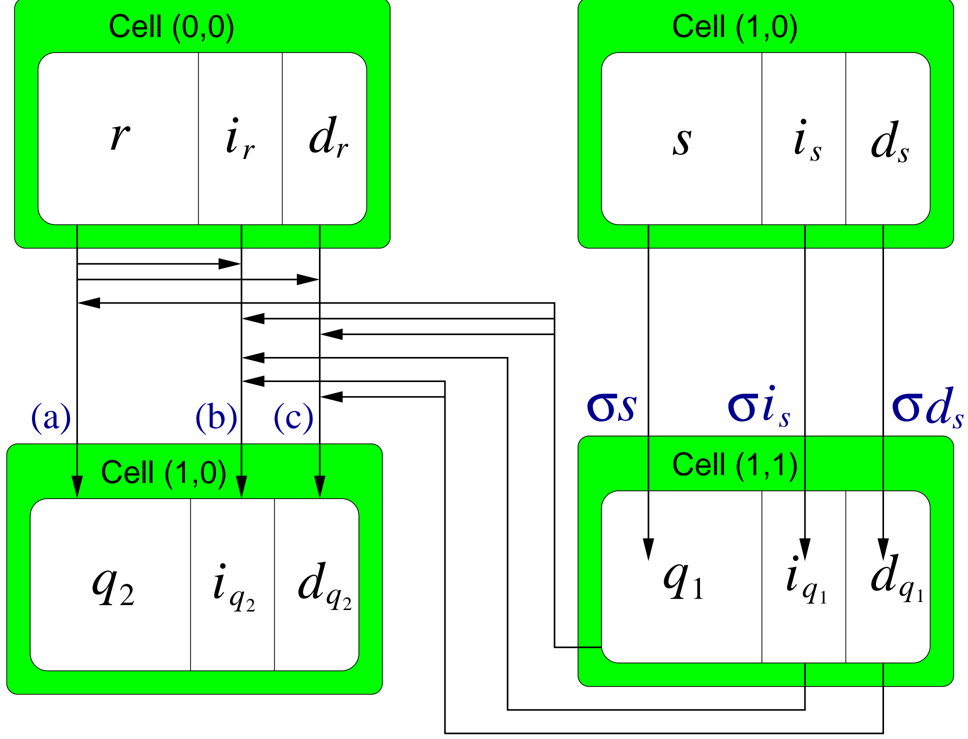


Figure 4.4: Update propagation data flow through cells in the ISS. The example shown is from Section 4.3, where the selection and join are separated out into different cells. The selection operation is $q_1 = \sigma s$, where σ is shorthand for $\sigma_{\tau_s[type]=1}$. The selection insert differential operation is $iq_1 = \sigma i_s$, and the selection delete differential operation is $dq_1 = \sigma d_s$. The join operation (a) is $q_2 = r \bowtie q_1$, where $\bowtie$ is shorthand for $\bowtie_{\text{Distance}(\tau_r[loc],\tau_s[loc])\leq 2}$. The join insert differential operation (b) is $i_{q_2} = ((((i_r \bowtie q_1) \uplus (r \bowtie i_{q_1})) \uplus (i_r \bowtie i_{q_1})) - (i_r \bowtie d_{q_1})) - (i_{q_1} \bowtie d_r)$, and the join delete differential operation (c) is $d_{q_2} = (((q_1 \bowtie d_r) \uplus (r \bowtie d_{q_1})) - (d_{q_1} \bowtie d_r))$.

page. If indexes can be used (e.g. indexes on join predicate attributes), then in the worst case, the number of disk page accesses is $\mathcal{O}(\log(n_r/M_r) + \log(n_l/M_l))$ where M_r (M_l) is the number of index entries for tuples in r (l) that will fit in a single disk page. Further analysis and proof for the join operation can be found in [32].

This approach will work for any query operation that distributes over $\uplus$ and $-$. This is the case for relational projection, selection, and joins. An example proof of the distributive property of the join operation over deletion can be found in the appendix of [32].

4.5 Pushing Data From Server to Client Using HTTP

After update propagation, and before applying the differential tables to the main tables, the spatial data of the non-empty differential tables are transmitted to the client for incremental update of the data visualization.

Pushing data from a server to a client in an Internet environment can present additional challenges when the client is behind a firewall. Many users may want to run their clients from behind a firewall for security reasons. Firewalls help prevent attack from outside by closing off communication ports through the firewall. A firewall configuration may allow clients to initiate connections through particular ports to servers on the outside. For example, port 80 is sometimes open for web browsers to access web servers outside the firewall. Opening ports so that connections may be initiated from outside the firewall is not desirable because it can make the system more vulnerable to attacks from outside.

Although access from clients inside a firewall to the outside may vary from firewall to firewall, many firewall administrators at least allow web browsing from inside the fire wall. This is done via the HTTP protocol through a specific port either directly or via a proxy. The Hypertext Transfer Protocol (HTTP) [21] is designed to pass Hypertext Markup Language (HTML) [8] documents between web servers and clients. HTTP tunneling is a technique used to pass arbitrary data back and forth between clients and servers by placing it in an HTTP wrapper, using the HTTP

protocol to send the data through the firewall.

The HTTP protocol enforces a client pull model. Every HTTP connection session has two phases. In the first phase, the client sends data to the server. In the second phase the server receives data from the client. Once the second phase starts, no more data may be sent to the server during that session. When the second phase is complete, the connection is closed. The protocol does not provide any way for servers to initiate connections to clients.

The HTTP client pull model is a problem in the ISS. Updates to the base relations on the server side from external sources other than the client require the server to push data to clients to update visualization displays. Since HTTP does not support server initiated transactions, the server can not initiate transactions to push data to the client using this protocol. To get around this, the client polls the server for updates in a continuous never ending HTTP session that quickly moves to the second phase and stays there. This is called the ISS *polling session*. If for some reason the connection is terminated (e.g., the proxy closes the connection after a time out period), then the client immediately establishes a new polling session.

Multi-threading can be used to enable the client to continually poll the server and still send data to the server at the same time allowing more than one HTTP connection or session to be active at a time between a client and server. Multi-threading is supported in most popular general purpose computer systems today, either directly by the operating system, or through programming libraries.

The threads handling the polling session are called the ISS *push threads*. There

is both a client side push thread and a server side push thread (see Figure 4.5). The client push thread initiates an HTTP polling session with the server. The server spawns its own server side push thread when contacted by the client to service the polling session. Other threads running in the server handle processing of data. The other threads push data to the client by placing the data on a queue. When data is placed on the queue, the server push thread wakes up, pops the data off the queue, and transmits it to the client. On the client side, the client push thread receives the data and places it on a queue. Another thread in the client, called the *data processing thread*, processes each item on the queue in turn. Having a separate thread to receive data and place it on the clients queue enables data to be transmitted while processing data already received in parallel.

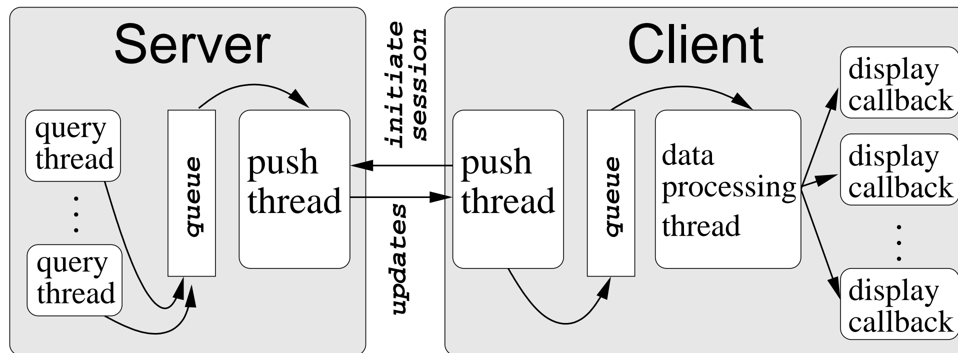


Figure 4.5: ISS client-server polling session. The client initiates the HTTP session. Updates generated in the server query threads are placed on a queue in the server. The server's push thread pops updates off the queue and transmits them to the client. The client push thread receives the updates and places them on the client's queue. The client data processing thread pops the data off the queue and executes the display callbacks. Each display callback is associated with a particular spreadsheet cell in the client.

If the data pulled off the queue is an update to a relation, or a query result, then the appropriate visualization displays are updated. To accomplish this, each cell controlling its own display window registers a callback with the communication event

thread to be invoked when data is received from the server. Callback procedures or methods are registered with the data processing thread to be invoked when particular kinds of data are transmitted from the server. The data processing thread invokes the appropriate callbacks depending on the nature of the data it pops off the queue.

Let us now consider again the example from Section 4.3 set up in the spreadsheet shown in Figure 4.3. Recall that relation s is updated at time t_0+1 by deleting tuple $\{y, (5,3), 1\}$ and inserting tuple $\{y, (4,3), 1\}$. The selection predicate on the view in cell (0,1) is $\tau_s[type] = 1$. The update propagation algorithm computes $d = \{y, (5,3), 1\}$, and $i = \{y, (4,3), 1\}$ for cell (0,1). Before moving on to the next cell, data from d and i are placed on the server's push queue by the query thread processing the transaction. The data needed for rendering which in this case is the object id and old location, $\{y, (5,3)\}$, and the object id and new location, $\{y, (4,3)\}$. The server's push thread reads this data off the queue, then transmits this information to the client where the client's push thread places the update information for cell (0,1) on its own queue. The data processing thread of the client in turn pops this data off the queue. It invokes all callbacks that are registered for update data on cell (0,1). These callbacks take the update information as an argument and use it to update data visualization displays.

4.6 Interactive Visualization

The spatial attributes of base relations and query results are transmitted to the client to be rendered. This enables interactive visualization without the need to transmit

data from the server again when the perspective changes and a new rendering is needed. The user can pan and zoom without any further interaction with the server.

In the ISS, every client cell owns its own display window to display spatial data from the corresponding cell on the server. The data used to create the rendering is saved in a separate data structure to be used for rendering when the perspective changes. Any cell can also display data belonging to other cells. This allows the user to overlay data from different cells in the same display window. Although this other data is owned by the other cells, it is not rendered by other cells. This is because each cell may have a different perspective (e.g. different zoom or pan). Each cell renders data according to its own perspective.

In a network based environment, the spatial data to be displayed must fit in main memory on the client. This requirement can be relaxed if the client is allowed to write files to the local machine. Non-spatial data, other than object ids, are not transmitted to the client except in small amounts when the user wants to know the details about a particular object.

4.7 Conclusion

As a proof-of-concept we implemented the algorithms and techniques described here in JAVA (client and server front end), Tcl (server interface), and C/C++ (database server engine). The implementation demonstrated the ability of this approach to achieve the goal of keeping client visualizations of spatial data up-to-date as materialized view results change. This was accomplished in the Internet environment

constrained by firewall security.

Chapter 5

K-Nearest Neighbor Queries

Consider the following queries. For a cell phone, keep track of the nearest cell tower. For a suspect getaway car, keep track of the nearest police cruiser. For a robot explorer, keep track of the nearest maintenance robot. For a ship, keep track of the nearest sonar tracking station. Each of these queries is an example of a nearest neighbor query that is maintained over time.

In this chapter, we address the maintenance of k-nearest neighbor queries on moving points. Most previous work in moving object databases has been in ad-hoc queries (e.g., where is the nearest police car right now), and future queries (e.g., which police cars will be the closest over the next 5 minutes given the current speed and direction of all cars). Little work has been done on the problem of query maintenance. For example, keeping track of which car is the nearest police car to a suspect vehicle, as they move around in real-time, and the database is updated to reflect changes in speed and direction.

Our main contributions in this chapter are algorithms based on an event-based query maintenance approach to maintain k-nn query results on continuously moving points over time. Although the examples given in this paper are 1-dimensional (1D) and show static query points, the techniques and algorithms are general and applicable to higher dimensions and moving query objects.

In this chapter, we expand on our work first presented in [35] to support updates to the query point, and to handle query circle underflow (see below for more details). Additionally, we present new experiments using both real and simulated data, whereas the experiments in [35] employed only synthetic data. Experimental results also include statistics on cpu intensive operations as well as disk accesses, whereas [35] reported only disk accesses in results.

In Section 5.1 we present our Continuous Windowing k-nn algorithm. In Section 5.2 we present an extension of other previous work to support updates. Performance issues of these algorithms are discussed in Section 5.3. Experimental results are given in Section 5.4.

5.1 Continuous Windowing KNN (CW)

In this section we present our approach, called the Continuous Windowing k-nn algorithm (CW). The CW algorithm is based on the observation that window queries are easier to maintain on moving points than k-nn queries, because w-events are fundamentally less expensive to process than oc-events (see Chapter 2). The CW algorithm filters points to be considered as nearest neighbor candidates using a within

query around the query point. If the within query selects at least k points, then only those points in the within query result need to be considered when computing the k-nn query.

Figure 5.1 is an example where the circle represents the circular query window. Conceptually, the data set is divided into two subsets, those that are close to the query point q called the *within set*, and those that are far away. Only points in the window are considered in searching for the next nn-event. For the example in Figure 5.1, points a, b, and g are the only points in the within set of q with a query window of radius r.

Figure 5.2 shows a 1D example with updates. This 1D example will be used to illustrate the algorithms below. Note that in these examples the query point is not moving, but the CW does support moving query points.

The CW algorithm is given in Figure 5.3. It makes use of the notation and functions defined in Chapter 2. Parameter r is a relation with a moving point Attribute, q is the query point, k is the number of neighbors that are sought, $x \geq 2$ is an integer used in determining the initial size of the window. We assume that parameter x is chosen to be sufficiently small to store $k + x + 1$ points well within the limits of main memory. Variable d is the radius of the within query window. Variable W is the within set. Variable Q is the w-event priority queue. Variable e_{nn} is the next nn-event, and K is the set of k nearest neighbors to q.

The algorithm starts by invoking procedure CW_Adjust_Window() (line 1). Procedure CW_Adjust_Window() computes the initial size of the query circle (radius d),

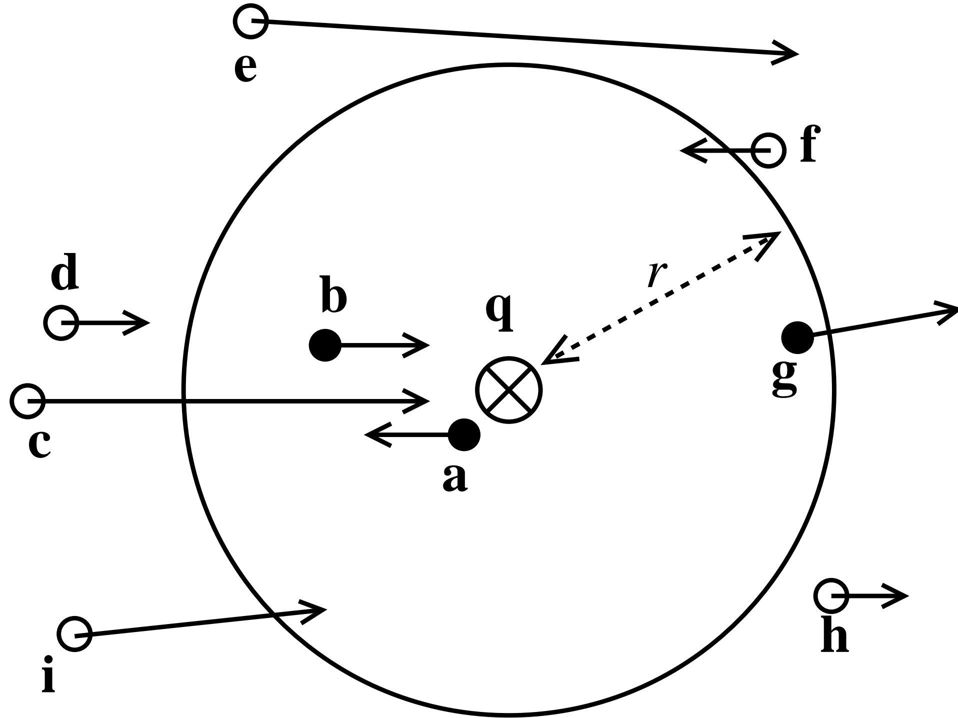


Figure 5.1: 2D example illustrating the CW approach, where $\otimes$ is the query point $\mathbf{q}$, r indicates the radius of the query window, $\bullet$ indicate points in $\mathbf{q}$'s within set, and $\circ$ indicate points not in the within set.

within set W, and w-events enqueued in priority queue Q sorted by time of the w-events (see Figure 5.4 for more details). Next in the CW algorithm, the within set W is used to compute the k-nn result set K, and the next nn-event e_{nn} in a call to CW_Compute_Knn_Result() (line 2) (see Figure 5.5 for more). The query result K is maintained indefinitely until it is no longer needed by the user (line 3). The algorithm does nothing until an update occurs or an event comes due (line 4). If there is an update to the data point relation r, then CW_Update_Data_Relation() (line 6) is invoked (see Figure 5.6 for more). If there is an update to the query point, then CW_Update_Query_Point() (line 8) is invoked (see Figure 5.7 for more). W-events are processed by CW_Process_Within_Evt() (line 10) (Figure 5.8), and nn-events are processed by CW_Process_Nn_Evt() (line 11) when they occur (Figure 5.9).

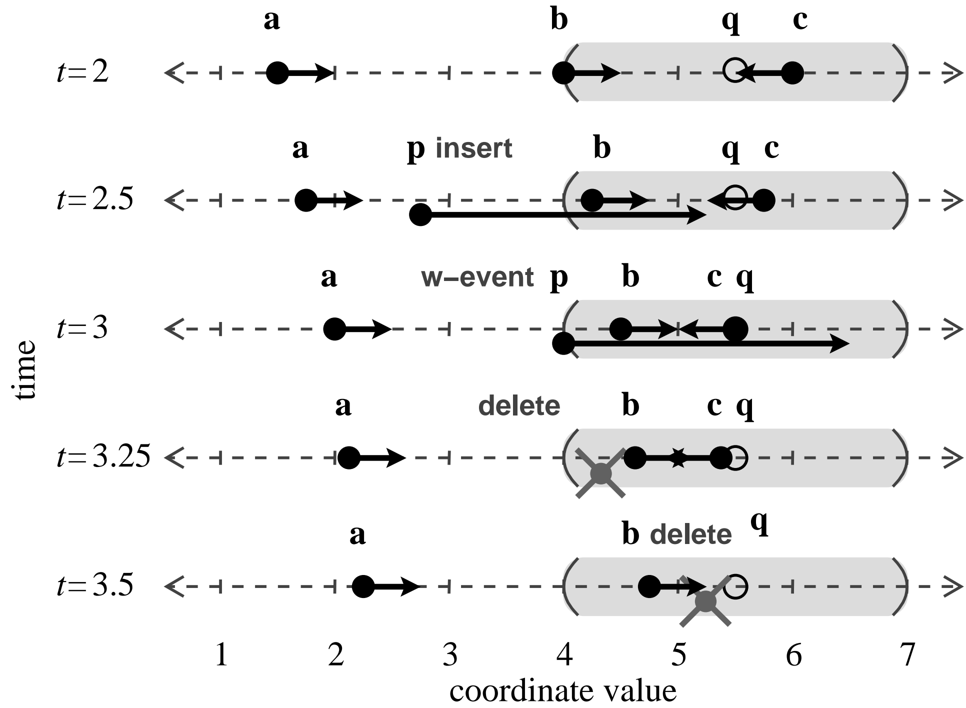


Figure 5.2: Example snapshots in time of 1D moving points, events, and updates up to time $t = 3.5$. Arrow length indicates distance traveled in one unit of time. The shaded area shows the extent of the query window within distance $d = 1.5$ of query point $\mathbf{q}$.

procedure Continuous_Windowing_Knn(r, q, k, x)

1. CW_Adjust_Window(r, q, k, x, d, W, Q)
2. CW_Compute_Knn_Result(q, k, e_{nn}, W, K)
3. **while** true **do**
4. Sleep until there is an update, or an event comes due.
5. **if** there is an update to r **then**
6. CW_Update_Data_Relation$(r, q, k, x, d, e_{nn}, W, K, Q)$
7. **else if** there is an update to q **then**
8. CW_Update_Query_Point$(r, q, k, x, d, e_{nn}, W, K, Q)$
9. **else if** a w-event has come due **then**
10. CW_Process_Within_Evt$(r, q, k, x, d, e_{nn}, W, K, Q)$
11. **else** CW_Process_Nn_Evt(q, e_{nn}, W, K)
12. **end while**

Figure 5.3: Continuous_Windowing_Knn() (CW)

Table 5.1 shows a trace of the CW algorithm for the example given in Figure 5.2 up through time $t = 3.5$. It illustrates how the updates in the example from Figure 5.2

can change the pending events for the CW algorithm. Column 1 indicates the current time for each row. Column 2 shows the update or event (if any) at time t. Column 3 shows the nearest neighbor (nn) at time t. Column 4 gives the next nn-event e_{nn}. Column 5 shows the w-events on Q at each time step t. At time $t = 2.5$, a tuple with moving point $\mathbf{p} = \mathsf{pt}(2.75, 2.5, 2.5)$ is inserted into relation r. Location $\mathbf{p}(2.5)$ is farther from $\mathbf{q}$ than $d = 1.5$ so a new w-event $\mathsf{w}(\mathbf{p}, 3)$ is added to the priority queue Q. At time $t = 3$, the w-event is processed and $\mathbf{p}$ is added to the within set W. Since the oc-event of $\mathbf{p}$ comes before the oc-event of any other point in W, it becomes the new nn-event $e_{nn} \leftarrow \mathsf{oc}(p, 3.5)$. At time $t = 3.25$, $\mathbf{p}$ is deleted, and the new nn-event, $e_{nn} \leftarrow \mathsf{oc}(b, 4)$, is computed by examining the remaining elements of $W = \{\mathbf{b}, \mathbf{c}\}$. At time $t = 3.5$, the nearest neighbor $\mathbf{c}$ is deleted.

t	update or event	nn	e_{nn}	Q
2	-	**c**	$\mathsf{oc}(\mathbf{b}, 4)$	$\langle \mathsf{w}(\mathbf{c}, 6), \mathsf{w}(\mathbf{a}, 7), \mathsf{w}(\mathbf{b}, 8) \rangle$
2.5	insert **p**	**c**	$\mathsf{oc}(\mathbf{b}, 4)$	$\langle \mathsf{w}(\mathbf{p}, 3), \mathsf{w}(\mathbf{c}, 6), \mathsf{w}(\mathbf{a}, 7), \mathsf{w}(\mathbf{b}, 8) \rangle$
3.0	w-event	**c**	$\mathsf{oc}(\mathbf{p}, 3.5)$	$\langle \mathsf{w}(\mathbf{p}, 4.2), \mathsf{w}(\mathbf{c}, 6), \mathsf{w}(\mathbf{a}, 7), \mathsf{w}(\mathbf{b}, 8) \rangle$
3.25	delete **p**	**c**	$\mathsf{oc}(\mathbf{b}, 4)$	$\langle \mathsf{w}(\mathbf{c}, 6), \mathsf{w}(\mathbf{a}, 7), \mathsf{w}(\mathbf{b}, 8) \rangle$
3.5	delete **c**	**b**	$\emptyset$	$\langle \mathsf{w}(\mathbf{a}, 7), \mathsf{w}(\mathbf{b}, 8) \rangle$

Table 5.1: A trace of the CW algorithm applied to the example in Figure 5.2.

CW_Adjust_Window(), given in Figure 5.4, computes the circular window by maintaining a set S of the $(k + x + 1)$ closest points from q as it scans the entire relation r (line 2). Set S is sorted by order of each point's current distance to query point q (line 3). Point $s_i \in S$ denotes the ith point in the ordered set. The distance from the query point to the edge of the circular window is the average distance from q to the furthest two points in the list (line 4). The last point is thrown out since its not in

the circle, and the remainder become the within set W (line 5). The next w-event for each point in the within set W is enqueued in Q (line 6–7). Recall from Section 2.5 that function $\mathsf{next_w_event}(p, q, d, t)$ returns the next event after time t when point p will be at distance d from q, or it returns a null event (denoted $\emptyset$) with time stamp ∞ if no such event exists.

```
procedure CW_Adjust_Window(r, q, k, x, d, W, Q)
1.  Let W ← ∅; Q ← ∅
2.  Scan r and find the closest k + x + 1 points to
    q at the current time, and assign them to set S.
3.  Sort points in S by distance to q at the current time.
4.  Let d ← (||q, s_{k+x},now|| + ||q, s_{k+x+1},now||)/2.
5.  Let W ← first k + x points in S.
6.  foreach point w_i ∈ W
7.     Q ←next_w_event(w_i, q, d,now)
```

Figure 5.4: CW_Adjust_Window()

CW_Compute_Knn_Result(), given in Figure 5.5, computes the k-nn result K given the within set W. We assume the size of W allows it to be stored stored and operated on entirely in main memory. All tuples in W are sorted by the distance of their moving point attribute from q at the current time. The first k tuples in the sorted list are assigned to set K (line 1)[1]. Procedure CW_Compute_Knn_Result() also computes the next nn-event e_{nn} from the points in W, with respect to the current k^{th} neighbor $\mathsf{Kth}(K)$ and query point q (lines 4–7). Function $\mathsf{Kth}(K)$ returns the kth nearest neighbor of K at the time K was last modified. Recall from Section 2.5 that function $\mathsf{next_oc_event}(p, q, nn, t)$ returns the next oc-event for point p after time

[1]For simplicity, and brevity, we do not consider the case when two moving points are at exactly the same distance from q at time t.

t with respect to query point q, and k^{th} neighbor nn.

```
    procedure CW_Compute_Knn_Result(q, k, e_nn, W, K)
1.  Let K be the closest k points to q in set W
    at the current time sorted by distance to q.
3.  e_nn ← ∅     /* note: Time(∅) = ∞ */
4.  foreach point w_i ∈ W ∧ w_i ≠Kth(K) do
5.     e_oc ←next_oc_event(w_i, q,Kth(K),now)
6.     if Time(e_oc) <Time(e_nn) then e_nn ← e_oc
7.  end foreach
```

Figure 5.5: CW_Compute_Knn_Result()

CW_Update_Data_Relation(), given in Figure 5.6, is invoked when relation r is updated. If there is no current query point, then nothing is done (line 1)[2]. When a point p is inserted (line 3), and falls within the query window (line 4), then the w-event for when it will exit the query window is computed and enqueued in Q (line 5), and p is added to the within set W (line 6). If the new point p is also closer to the query point q than the current k^{th} neighbor (line 7), then it pushes out the kth neighbor from the query result, i.e., the kth neighbor becomes the $(k+1)^{\text{th}}$ neighbor. This results in the kth neighbor being removed from the result set K, and the new point added (line 8). Since the kth neighbor has now changed, a new nn-event is computed (line 9). If the new point p is in the within set W, but is not in the query result, then its checked to see if it will become the new k^{th} neighbor before any other point in the within set (lines 11-12). If a point is inserted, but does not fall inside the query window, then its enter event is enqueued, if one exists (line

[2]We assume that a query point deletion is shortly followed by a query point insertion, but not necessarily at the exact same time step.

14).

```
    procedure CW_Update_Data_Relation(r, q, k, x, d, e_nn, W, K, Q)
1.  if there is no current query point then return.
2.  Let point p be the point just inserted or deleted from r.
3.  if p was inserted into r then
4.     if ||q, p,now|| ≤ d then
5.        Let Q ← Q ∪ next_w_event(p, q, d,now).
6.        Let W ← W ∪ p.
7.        if ||q, p,now|| < ||q,Kth(K),now|| then
8.           Let K ← (K−Kth(K)) ∪ p.
9.           Let e_nn ← soonest oc-event from points in W
             after current time.
10.       else
11.          Let e_oc ←next_oc_event(p, q,Kth(K),now)
12.          if Time(e_oc) <Time(e_nn) then e_nn ← e_oc
13.       end if-then-else
14.    else if next_w_event(p, q, d,now) < ∞ then
15.       Let Q ← Q ∪ next_w_event(p, q, d,now).
16.    end if-then-else
17. else if p has a w-event in Q then
18.    Remove the w-event involving p from Q.
19.    if p is in W then
20.       Let W ← W − p.
21.       if |W| ≤ k then /* underflow */
22.          CW_Adjust_Window(r, q, k, x, d, W, Q)
23.          CW_Compute_Knn_Result(q, k, e_nn, W, K)
24.       else if p in K then
25.          CW_Compute_Knn_Result(q, k, e_nn, W, K)
26.       else if p involved in e_nn then
27.          Let e_nn ← soonest oc-event from points in W
             after current time.
28.       end if-then-else
29.    end if
30. end if-then-else
```

Figure 5.6: CW_Update_Data_Relation()

When point p is deleted, and has a w-event in Q (line 17), then its w-event is removed from Q (line 18). Additionally, if p is a member of the within set W (line

19), then it is removed from set W (line 20). If removing the point from set W results in query window underflow (line 21), then the query window needs to be expanded to include more points (line 22) , and the query result and the nn-event are recomputed (line 23). If underflow does not occur, but the deleted point p is in the result set K (line 24), then the query result and nn-event are recomputed because the kth neighbor will have changed (line 25). If underflow does not occur, and the deleted point p is not in the result set K, but p is involved in the current nn-event (line 26), then the nn-event is recomputed (line 27).

CW_Update_Query_Point() given in Figure 5.7, is invoked when the query point is updated. When the query point q is inserted (line 1), then a new query window, within set, w-events (line 2), result set K, and next nn-event (line 3) are computed. When query point q is deleted, then the query result and all events are removed (line 4). Note that there is at most one query point at any given time.

procedure CW_Update_Query_Point($r, q, k, x, d, e_{nn}, W, K, Q$)
1. **if** query point q was inserted **then**
2. CW_Adjust_Window(r, q, k, x, d, W, Q)
3. CW_Compute_Knn_Result(q, k, e_{nn}, W, K)
4. **else** let $K \leftarrow \emptyset, Q \leftarrow \emptyset, e_{nn} \leftarrow \emptyset$.

Figure 5.7: CW_Update_Query_Point()

CW_Process_Within_Evt() in Figure 5.8 is invoked on w-events. Recall from Section 2.2, an *enter* event is a w-event where the data point is moving closer to the query point at event time, and an *exit* event is a w-event where the data point is moving away from the query point at event time. In CW_Process_Within_Evt(), if the

next w-event on the queue is an enter event (line 3), then the data point p is added to within set W (line 4), the exit event for p is enqueued (line 5), and p's oc-event event is considered for the next nn-event (lines 6–7). If the w-event is an exit event (line 8), then the data point p of the w-event is removed from within set W (line 9). If the exit of point p from the query window results in underflow (line 10), then the window is resized to include more data points, sets W and K are recalculated given the new window, and new events are computed (lines 11–12). If no underflow occurs, but p is involved with the next nn-event (line 13), then a new nn-event is computed (line 14).

```
procedure CW_Process_Within_Evt(r, q, k, x, d, e_nn, W, K, Q)
 1. e_w ←Pop(Q).
 2. Let p be the data point involved in e_w.
 3. if e_w is an enter event then
 4.    Let W ← W ∪ p.
 5.    Let Q ← Q ∪ next_w_event(p, q, d,now).
 6.    Let e_oc ←next_oc_event(p, q,Kth(K),now).
 7.    if Time(e_oc) <Time(e_nn) then e_nn ← e_oc.
 8. else /* e_w is an exit event */
 9.    W ← W − p.
10.    if |W| ≤ k then /* underflow */
11.       CW_Adjust_Window(r, q, k, x, d, W, Q)
12.       CW_Compute_Knn_Result(q, k, e_nn, W, K)
13.    else if p is involved in event e_nn then
14.       Let e_nn ← soonest oc-event from points in W
          after current time.
15.    end if-then-else
16. end if-then-else
```

Figure 5.8: CW_Process_Within_Evt()

CW_Process_Nn_Evt(), given in Figure 5.9, is invoked on nn-events. Of the two data points involved in nn-event e_{nn}, if the data point p that is not currently the

k^{th} neighbor is moving to be closer to the query point than the kth neighbor after the event (line 2), then the query result is updated since p will push out the old k^{th} neighbor from the query result K (line 3). In any case, the kth neighbor has changes, so a new nn-event is computed with respect to the new k^{th} neighbor (line 4).

procedure CW_Process_Nn_Evt(q, e_{nn}, W, K)
1. Let $p \leftarrow$ non-k^{th} neighbor data point involved in e_{nn}.
2. **if** p will be closer to q than Kth(K) after event **then**
3. $\quad K \leftarrow (K - Kth(K)) \cup p$.
4. Let $e_{nn} \leftarrow$ soonest oc-event from points in W after current time.

Figure 5.9: CW_Process_Nn_Evt()

5.2 Extending TP KNN for Updates

In this section we extend the continuous TP KNN algorithm presented in [65] to support updates during processing of the query, termed the *extended TP* (ETP) algorithm. This algorithm is compared experimentally with CW in Section 5.4. All notation, variables and functions are as described in the previous sections unless otherwise specified.

Extended_TP_Knn() (ETP), given in Figure 5.10, is similar to the top-level CW algorithm presented in Figure 5.3. The main loop of Extended_TP_Knn() is nearly identical to the main loop of Continuous_Windowing_Knn() presented in Figure 5.3, except for the lack of a subroutine to process w-events since their are no w-events

used in the ETP approach. The primary difference between the CW algorithm and the ETP algorithm is that the ETP algorithm uses a TPR-tree to find members of the k-nn query result set and nn-events. The CW algorithm, on the other hand uses the within set to find the k-nearest neighbors and nn-events. Consequently, there are no w-events processed in the ETP approach. In the figure, parameter r is a relation with a moving point attribute in its schema, q is the query point, and k is the number of neighbors that are sought. Variable e_{nn} is the next nn-event. Variable tpr is a TPR-tree index on the moving point attribute in r. Variable K is the set of k neighbors of q.

The first step is to build a TPR-tree index (see Section 3.7.1) on the moving points in relation r (line 1). The index is used in an *incremental distance query* to find the closest k neighbors of query point q (line 2). This is the same *incremental distance query* algorithm presented in [30] (see Section 3.6). The TPR-tree index is also used in a *next nn-event query* to find the next nn-event (line 3). The same method presented in [65] is used here to perform the *next nn-event query* here (see Section 3.7.3).

Table 5.2 shows a trace of the ETP algorithm for the example given in Figure 5.2 up through time $t = 3.5$. Column 1 indicates the current time for each row. Column 2 shows the update or event (if any) at time t. Column 3 shows the nearest neighbor (nn) at time t. Column 4 gives the next nn-event e_{nn}. Only one event is computed at a time, so there is no event queue as for the CW algorithm. At time $t = 2.5$, point $\mathbf{p} = \mathsf{pt}(2.75, 2.5, 2.5)$ is inserted. The oc-event for the new point $\mathbf{p}$ comes before the

```
procedure Extended_TP_Knn(r, q, k)
 1.  Build TPR-tree index, tpr on the moving points in r.
 2.  Let K ← first k points returned by the incremental
     distance query on index tpr for query point q.
 3.  Let e_nn ← result of next nn-event query on index tpr
     for query point q and k^th neighbor Kth(K).
 4.  while true do
 5.     Sleep until there is an update, or event e_nn comes due.
 6.     if there is an update to r then
 7.        ETP_Update_Data_Relation(r, q, k, e_nn, tpr, K)
 8.     else if there is an update to q then
 9.        ETP_Update_Query_Point(q, k, e_nn, tpr, K)
10.     else ETP_Process_Nn_Evt(q, k, e_nn, tpr, K)
11.  end while
```

Figure 5.10: Extended_TP_Knn() (ETP)

oc-event of any other point in the entire data set, so it becomes the new nn-event $e_{nn} \leftarrow \mathsf{oc}(p, 3.5)$. Point **p** is deleted at time $t = 3.25$. This means the oc-events of the points in the data set must be examined again to find the next to occur. In this case, the oc-event for **b** becomes the next nn-event. When the current nearest neighbor **c** is deleted at time $t = 3.5$, point **b** becomes the new nearest neighbor. The only other point in the data set now is point **a** with nn-event $e_{nn} = \mathsf{oc}(\mathbf{a}, 7.5)$.

t	update or event	nn	e_{nn}
2	-	**c**	$\mathsf{oc}(\mathbf{b}, 4)$
2.5	insert **p**	**c**	$\mathsf{oc}(\mathbf{p}, 3.5)$
3.25	delete **p**	**c**	$\mathsf{oc}(\mathbf{b}, 4)$
3.5	delete **c**	**b**	$\mathsf{oc}(\mathbf{a}, 7.5)$

Table 5.2: ETP trace for the example in Figure 5.2.

ETP_Update_Data_Relation(), given in Figure 5.11, is invoked on updates to data

relation r. When point p is inserted into relation r (line 2), it is also inserted into the TPR-tree index (line 3). If there is no current query point, then nothing else need be done (line 4). If there is a query point, and p is closer to it than the current k^{th} neighbor (line 5)[3] , then p is added to the result set, and the old k^{th} neighbor is removed (line 6). Since the kth neighbor has changed, the new nn-event is found using the *next nn-event query* (see Section 3.7.3) on the TPR-tree index (line 7). If p is not in the result set, then its oc-event is checked against the current nn-event e_{nn}. If p's oc-event occurs sooner than e_{nn} then the oc-event becomes the new nn-event (lines 9–10).

When point p is deleted from relation r, it is removed from the TPR-tree index (line 13). If no query point exists, then nothing else is done (line 14), otherwise if p was in the query result set (line 15), then the new k^{th} neighbor p_k is found using the *incremental distance query* algorithm (see Section 3.6) on index *tpr* (line 16). Point p is then removed from the query result set K, and the new k^{th} neighbor p_k is added to K (line 17). Since the kth neighbor has changed, a new nn-event for the the kth neighbor is found (line 18). If p is not in the query result set, then the current nn-event e_{nn} is checked to see if p is involved in e_{nn} (line 19). If it is, then a new nn-event is computed (line 20).

ETP_Update_Query_Point(), given in Figure 5.12, is invoked on updates to the query point. If a query point q is inserted (line 1), then the query result set K is found using the *incremental distance query* algorithm (line 2), and the next nn-event

[3]For simplicity, we do not consider the case where p and the kth neighbor are at exactly the same distance to q.

procedure ETP_Update_Data_Relation$(r, q, k, e_{nn}, tpr, K)$

1. Let p be the point just inserted or deleted in r.
2. **if** p was inserted into r **then**
3. Insert p into index tpr.
4. **if** there is no current query point **then** return.
5. **if** $||q, p,\text{now}|| < ||q,\text{Kth}(K),\text{now}||$ **then**
6. $K \leftarrow (K - \text{Kth}(K)) \cup p$.
7. Let $e_{nn} \leftarrow$ result of *next nn-event query* on index tpr for query point q and new $\text{Kth}(K)$.
8. **else** /* $\text{Kth}(K)$ closer to q than p */
9. Let $e_{oc} \leftarrow \text{next_oc_event}(p, q, \text{Kth}(K), \text{now})$.
10. **if** $\text{Time}(e_{oc}) < \text{Time}(e_{nn})$ **then** $e_{nn} \leftarrow e_{oc}$.
11. **end if-then-else**
12. **else** /* p was deleted from r */
13. Remove p from index tpr.
14. **if** there is no current query point **then** return.
15. **if** $||q, p,\text{now}|| < ||q,\text{Kth}(K),\text{now}||$ **then**
16. Let $p_k \leftarrow k^{th}$ point returned by *incremental distance query* on index tpr for query point q.
17. $K \leftarrow (K - p) \cup p_k$.
18. Let $e_{nn} \leftarrow$ result of *next nn-event query* on index tpr for query point q and new $\text{Kth}(K)$.
19. **else if** p is involved in event e_{nn} **then**
20. Let $e_{nn} \leftarrow$ result of *next nn-event query* on index tpr for query point q and new $\text{Kth}(K)$.
21. **end if-then-else**
22. **end if-then-else**

Figure 5.11: ETP_Update_Data_Relation()

e_{nn} is found using the *next nn-event query* algorithm (line 3). If q is deleted, then the query result (line 5), and the nn-event are removed (line 6).

ETP_Process_Nn_Evt(), given in Figure 5.13, is invoked when the nn-event e_{nn} comes due. Point p is the data point involved in nn-event e_{nn} that is not the current k^{th} neighbor of the query point (line 1). If p is not already in the query result set K (line 2), then it is added to the result set K and the old k^{th} neighbor is removed (line

procedure $\mathsf{ETP_Update_Query_Point}(q, k, e_{nn}, tpr, K)$
1. **if** inserting q **then**
2. Let $K \leftarrow$ first k points returned by *incremental distance query* on index *tpr* for query point q.
3. Let $e_{nn} \leftarrow$ result of *next nn-event query* on index *tpr* for query point q and $\mathsf{Kth}(K)$.
4. **else** /* deleting q */
5. $K \leftarrow \emptyset$.
6. $e_{nn} \leftarrow \emptyset$.
7. **end if-then-else**

Figure 5.12: ETP_Update_Query_Point()

3). One of the other points in K is now the new k^{th} neighbor. The new k^{th} neighbor is used along with the query point to find the next nn-event by invoking the *next nn-event query* algorithm (line 4).

procedure $\mathsf{ETP_Process_Nn_Evt}(q, k, e_{nn}, tpr, K)$
1. Let $p \leftarrow$ non-k^{th} neighbor data point involved in e_{nn}.
2. **if** p will be closer to q than $\mathsf{Kth}(K)$ after the event occurs **then**
3. $K \leftarrow (K - Kth(K)) \cup p$.
4. Let $e_{nn} \leftarrow$ result of *next nn-event query* on index *tpr* for query point q and k^{th}neighbor $\mathsf{Kth}(K)$.

Figure 5.13: ETP_Process_Nn_Evt()

5.3 Performance Issues

Analysis of algorithms for kinematic data is difficult without making many simplifying assumptions. Performance is dependent on many factors such as data set size, location distribution, velocity distribution, distribution of updates among tuples, and update frequency distribution. Instead of attempting a rigorous analysis on an overly

constrained subset of these factors, this section discusses some key performance issues of the ETP, CW, and Plane-Sweeping technique (PS) (described in Section 3.7.2), and how these factors play a part in the performance of each algorithm.

We assume that for large data sets, the majority of the data is stored on disk. Accesses to disk are orders of magnitude slower than memory, so cost is measured in number of disk accesses. For the sake of discussion, assume that all moving point data and query points share the same location, velocity, and update rate distributions. Ignoring esoteric cases, assume that all points are moving relative to the query point, and that they are not all moving in the same direction and at the same speed. Note that no implementation details are given for the PS method presented in [49], and thus we need to make assumptions for this approach in order to analyze it. We assume that PS creates a sorted list L_{PS} of all points by distance to the query point, and that is makes use of an event queue Q_{PS}. Let us assume an implementation using B-tree variants for both L_{PS}, and Q_{PS} to support efficient insertions and deletions. The CW event priority queue Q_{CW} is implemented using a B+-tree variant (see Section 5.4).

Initial Build: All three methods require an initial scan of some relation r. ETP scans r to build the TPR-tree index. CW scans the relation to find the query result and pending w-events. PS creates a list sorted by distance.

Data Structure Size: Let n be the size of the point data set. The asymptotic upper bound for the TPR-tree, L_{PS} (ignoring Q_{PS} for now), and Q_{CW} data structures is $\mathcal{O}(n)$. The lower bounds for each data structure are not the same. The entire data

set must be inserted into the TPR-tree, and the sorted list L_{PS}, giving a lower bound of $\Omega(n)$. The best case for CW is when no points outside the within query result will enter the within query window in the future. In this case, the only points involved in events in Q_{CW} are those in the within query result giving a lower bound of $\Omega(|W|)$. Given our assumption that $|W| \ll n$, it is likely that the size of Q_{CW} will be much smaller than the data structures for the other approaches.

Rebuilds: Rebuilding these structures from scratch may be required on occasion. Let UI be the average time period between two updates for a single object. By experimentation, Saltenis et. al. [58] determined that the performance of the TPR-tree degrades after time UI because almost all the entries have been updated by that time thereby causing the index to degenerate due to increasing overlap of the index nodes. They conclude the TPR-tree should be rebuilt when time UI is reached. The PS priority queue Q_{PS} needs to be rebuilt whenever the query point is updated because all the events on the queue will no longer be valid. The expected time between updates to the query point is also UI if we assume the same update rate distribution for the query point as the rest of the data set. A rebuild is also needed for CW when the query point is updated or the within set W underflows. The failure rate for these constraints depends heavily on the characteristics of the data and the method used to determine d. At the least, we would expect the CW method to rebuild more often than the other methods.

Number of Events: Only the nn-events are processed in the ETP approach. This makes the ETP approach optimal in the number of events processed throughout the

course of a query. There is only one event pending at any one time. CW processes additional w-events. The number of w-events over the course of a query, or on the event queue at any one time, depends on the selectivity of the within window and the motion characteristics of the data. PS processes an oc-event every time a neighbor changes position in L_{PS}. This includes the nn-event. The number of events on Q_{PS} at any one time is $\mathcal{O}(n)$. Since these events occur when points change order in L_{PS}, it is easy to imagine cases when the distances between neighbors on the list are small and thus many of these events on the queue will be imminent (e.g., points moving with different speeds and directions). In such cases, many more events would be processed over the course of a query than what the CW method would require.

Cost of Events: The cost of processing each event for each method is not the same. For the PS method it is only necessary to examine the immediate neighbors of points that switch order on L_{PS} to find the next time they will switch order with their new neighbors. Assuming a B-tree structure for L_{PS} yields a cost of at most $\mathcal{O}(\log n)$ to find the neighbors. The cost of event updates in Q_{PS} is also $\mathcal{O}(\log n)$. The CW approach is even cheaper requiring no disk accesses to examine other points when either a w-event or oc-event is processed because all the points that need to be examined are already in main memory. The only cost is in updating the event queue which is $\mathcal{O}(\log n)$. In the ETP approach, the cost of processing an event is $\mathcal{O}(n)$ for the *incremental distance query*. The worst case for the *incremental distance query* happens when all points are at the same distance from the query point. In practice this is unlikely in low dimensional data sets. The ETP method has no event queue.

The entire cost of the ETP method lies in the TPR-tree operations.

5.4 Experimental Results

This section presents experimental results comparing the ETP (Section 5.2) and CW (Section 5.1) algorithms. Our primary metric for cost is the number of disk accesses needed to compute and maintain a query. This is because accessing data on disk is several orders of magnitude slower than accessing data in memory.

We did not implement the PS algorithm since the approach is theoretical and no implementation details were given in [49]. In any case, from our analysis in Section 5.3, it appears that the PS data structures will be large (i.e., $\mathcal{O}(n)$). It also appears that the frequency of the occurrence of events would likely be higher than in the CW or ETP approaches, since the differences in distances between points on the list will be small, especially in the case of uniform data.

We use code provided by Saltenis et. al. from their original implementation of the TPR-tree [58][4]. This was built on the GiST [28] code version 0.9beta1. We extended this with implementations of the *incremental distance query* [30], and the *next nn-event query* [65]. In [35], we did not find a significant difference between *depth first* and *best first* versions of the TPR-tree index search algorithms (i.e, incremental distance query, next nn-event query), (see [35] for more details) so we use only *best first* versions in the experiments presented here.

[4] A special thanks to Saltenis et. al. and Tao et. al. for making their code available for use and study.

To implement the CW priority queue, we use a B+-tree variant of a priority search tree called the *Event B-tree* (EB-tree). In our implementation, every point has a unique id, or key. The priority queue is a B+-tree ordered by key to support efficient insertions and deletions of events. In addition to propagating the min-max key up the B+-tree, the earliest event time of all events in each subtree is also propagated up to the root. The earliest event in the tree is found by following the minimum event time down the branches of the tree to the leaf in which it is stored. The time of the next event can be found by examining just the root node. Figure 5.14 shows an example EB-tree. Our EB-tree is implemented using the same GiST code used by Saltenis et. al [58].

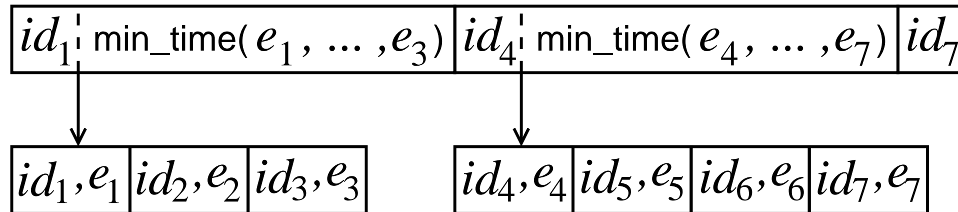


Figure 5.14: Example EB-tree with one root node, and two leaf nodes.

5.5 Data Sets

We used both real aircraft flight data and synthetic uniformly-distributed data in our experiments. Data sets consist of an initial set of moving points described as a linear function of time ($p(t) = \overrightarrow{x_0} + (t - t_0)\overrightarrow{v}$), and updates to the function coefficients ($\overrightarrow{x_0}$, t_0, $\overrightarrow{v}$) over time. A data set is characterized by the mean and standard deviation in the number of moving points (cardinality) at any given time, the period of time covered by the data set, and the average update interval. The average update interval

(UI) is the average length of time between updates for any given point.

All synthetic uniformly-distributed data sets are generated using a data generation tool developed by Saltenis et. al. [58]. The synthetic moving points are uniformly distributed over a 1000x1000 coordinate space. The speed of each point is uniformly distributed between 0 and $3/60 = 0.05$ coordinate distance units per time unit. All synthetic moving points are inserted at the start time of the dataset. Updates change the speed, but not the current location of each point. No new points are introduced after the start time, nor are any removed. The average update interval (UI) for our synthetic data is 600 time units. Each synthetic data set covers 3600 time units. The UI and speed relative to the size of the coordinate space of the synthetic data were chosen to be similar to the aircraft flight data for comparability.

Real commercial aircraft flight data was acquired as location data sampled at one minute intervals. Figure 5.15 shows an example snapshot in time to see how the data is clustered. The latitude-longitude of sampled locations were converted to linear functions describing aircraft motion by first applying the Douglas-Peucker line simplification algorithm [18] to the 2D latitude-longitude points forming a polyline from earliest to latest sampled location in time.[5] In our application of the Douglas-Peucker algorithm, we used a maximum error bound of $0.0\overline{6}$ degrees. Distortions introduced by the latitude-longitude projection onto the Earth's surface was ignored. The resulting vertices serve as the start locations for each update. Each vertex has an associated time stamp. The line segment to the next vertex divided by the time

[5]Although experiments were conducted on 2-dimensional data, the algorithms presented in this paper are applicable to higher dimensions.

difference between their time stamps gives the velocity vector for each update. The result was an average update interval of 700–735 seconds. The aircraft data sets cover a window $[20°, 60°]$ latitude by $[-135°, -60°]$ longitude. Since only about 5000 aircraft are in the air at any one time, larger data sets are generated by combining flights on different days during the same time period. Each aircraft data set covers a time period of two hours.

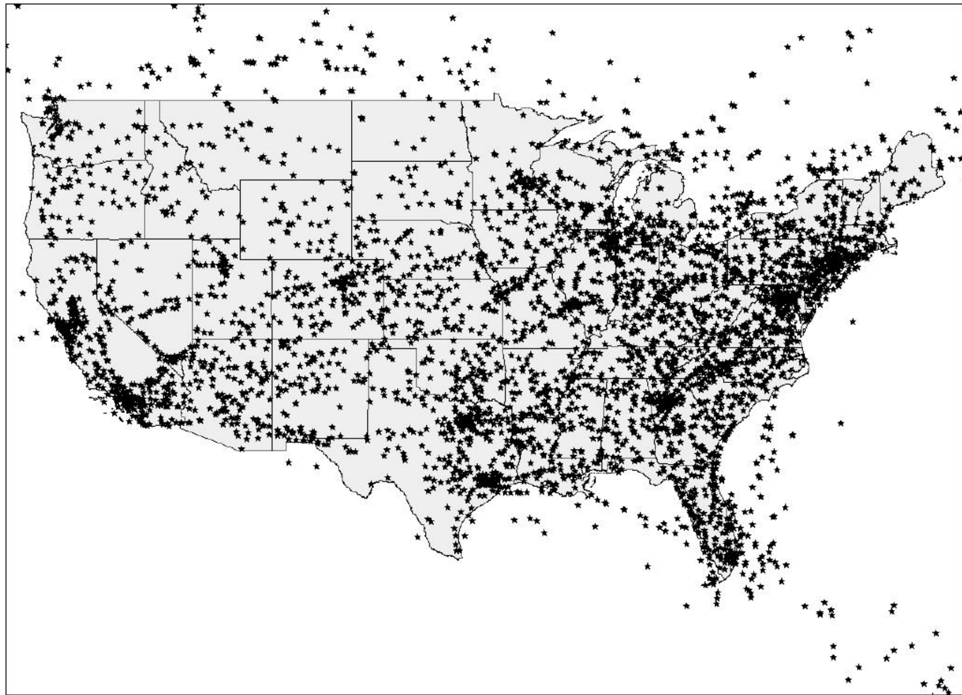


Figure 5.15: Snapshot of aircraft flight data.

One significant difference between the real and synthetic data is in the size of the data set at any given time. The number of points for the synthetic data stays constant, but the real flight data changes in the number of aircraft as flights land and take off.

Table 5.3 shows statistics for the data sets used in the experiments of this chapter. The table shows the mean and standard deviation in the size of the data sets over

the entire 2 hour time interval covered by each data set. The figure also shows the average update interval (UI) for each aircraft data set.

μ	9021	21020	29551	41855	50822
σ	680.8	1591	2223	3386	4381
UI	712.8	729.0	724.6	727.5	726.6

Table 5.3: Each column corresponds to a different aircraft data set. Each row is a statistic on the data sets. Row 1 is the mean number of flights at any given time (μ). Row 2 is the standard deviation in the number of flights (σ). Row 3 is the average update interval (UI) in seconds.

To make full use of the real data sets available, each data set was divided into 12 subsets starting at evenly spaced start times over the duration of the data set. For example, for a data set covering a time period of 900 units, the subset start times are 0, 75, 150, etc. If the duration of the experiment is longer than the time between subsets, then the subsets overlapped. If needed, the spacing between start times was adjusted so the experiments didn't run past the time covered by the subset. For our 900 time unit example above, if the experiment duration is 100 time units, then the time between start times might be only 72 time units. The time domains of each subset were then transformed to start at time 0.

5.6 Experimental Results

Default parameter values for each experiment, unless otherwise specified in the description of an individual experiment, are as follows. Duration of each experiment is 1000 time units (the time duration in [35] was only 60 time units). Disk page size is 4096 bytes. Number of neighbors $k = 1$. Experiments that do not vary by data

set size are run on a data set of 50000 points for uniform data, and an average of 50822 points for aircraft data. The number of pages in the cache for the TPR-tree index *tpr* is 64 pages. The event queue disk cache is 8 pages. Each cache uses a least recently used (LRU) page replacement policy. For the CW algorithm, the number of extra points to find (parameter x in Figure 5.3) is $x = 4$. Disk accesses are computed as an average over 100 experiments for a given set of parameters. All results report statistics accumulated after the initial loading data structures, and flushing of the disk based data structure caches.

For the CW algorithm, whenever there is a query point update or an underflow of the within set, the base relation is scanned. We assume one page access for every 93 moving point objects in the data set at the time the relation is scanned. This number was derived as follows. A 2D kinematic point is represented by 5 double floating point numbers of 8 bytes each, two doubles for the start location coordinates, two for the velocity vector, and one for the start time. Each moving point also has a unique identifier represented by a 4 byte integer. This gives a total of $(5 * 8 \text{ bytes}) + 4$ bytes $= 44$ bytes per object. Each page is 4096 bytes, therefor $\lfloor 4096$ bytes per page/44 bytes per object$\rfloor = 93$ objects per page.

The purpose of the first experiment is to determine which algorithm, CW or ETP, performs better in terms of disk accesses for different data set sizes. The results are given in Figures 5.16 and 5.17 for aircraft and uniform data respectively. In both figures, the x-axis is the average data set size, and the y-axis is the number of disk accesses in thousands (k). The points indicated by $\triangle$ symbols are the number of

disk accesses for the CW algorithms, while the $\diamond$ symbols indicates the number of disk accesses for the ETP algorithm. For the aircraft data set, the CW algorithm has over 17 times fewer disk accesses than the ETP algorithm for the largest data sets tested. For the uniform synthetic data, the CW algorithm has over 29 times fewer disk accesses than the ETP algorithm for the largest data sets tested.

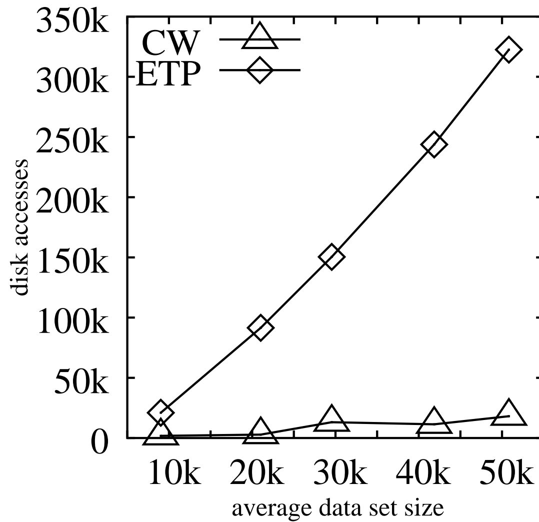


Figure 5.16: Disk accesses vs. data set size for aircraft data.

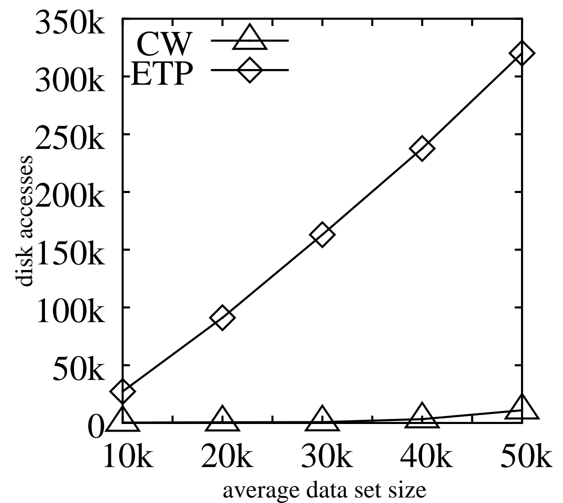


Figure 5.17: Disk accesses vs. data set size for uniform data.

An additional experiment was conducted to show relative performance of the two algorithms when there are no updates. The results are given in Figure 5.18 for both aircraft data set of approximately 50k points, and uniform data set size of 50k points. The results were obtained by simply ignoring all subsequent updates once the experiment started and processing events only. Two pairs of vertical bars are shown for each data set. The black bar on the left indicates the number of disk accesses for the ETP algorithm, and the white bar on the right indicates the number of disk accesses for the CW algorithm. When there are no updates, the ETP algorithm has 72 time fewer disk accesses than the CW algorithm for aircraft data, and 57 times

fewer disk accesses for uniform data. However, the number of disk accesses even for the CW algorithm, is relatively small (< 1000).

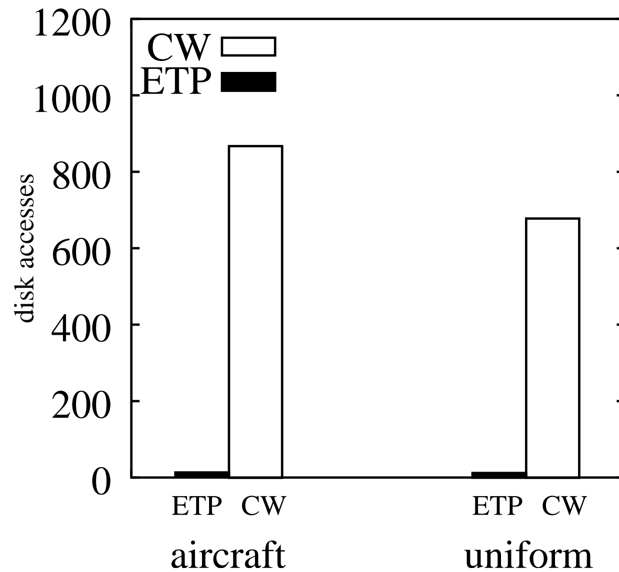


Figure 5.18: No updates

The performance shown in Figures 5.16 and 5.17 likely correlates with size of the disk based data structures. Figures 5.19 and 5.20 support this hypothesis. The y-axis is the number of entries in thousands (k) that are stored in each data structure (TPR-tree or event queue). The x-axis is the average data set size. Disk access for the CW algorithms are indicated by the $\triangle$ symbols, and disk accesses for the ETP algorithm are indicated by the $\diamond$ symbols. For the largest data set, the CW event queue has 33 times fewer entries than the ETP TPR-tree index for the aircraft data, and 96 times fewer entries for the uniform data.

The purpose of the next experiment is to determine how queries for more neighbors affect performance of the two algorithms. The results are given in Figures 5.21 and 5.22 for aircraft and uniform data respectively. The number of disk accesses are plotted on the y-axis versus the number of neighbors (k) along the x-axis. Disk ac-

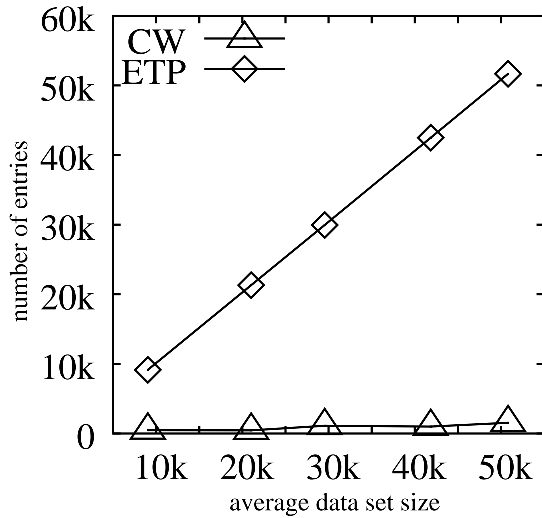


Figure 5.19: Number of entries vs. data set size for aircraft data.

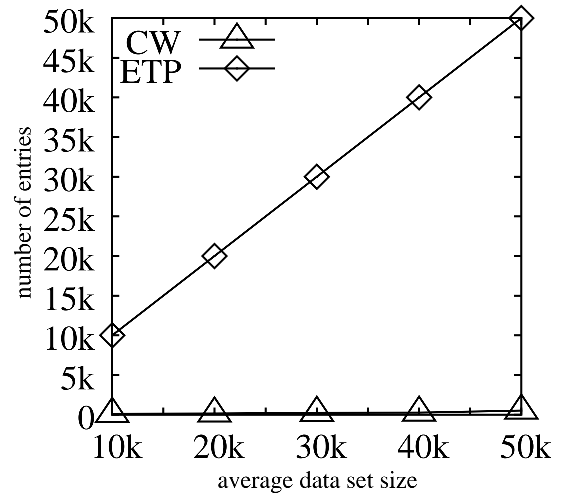


Figure 5.20: Number of entries vs. data set size for uniform data.

cess for the CW algorithms are indicated by the △ symbols, and disk accesses for the ETP algorithm are indicated by the ◇ symbols. The results show that an increase in the number of neighbors has a relatively small affect on overall performance.

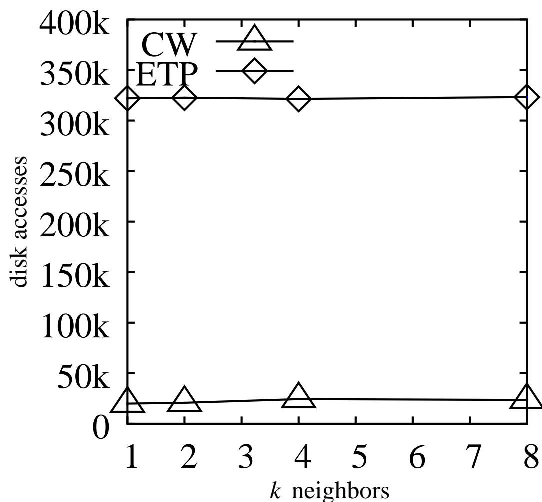


Figure 5.21: Disk accesses vs. number of neighbors for aircraft data.

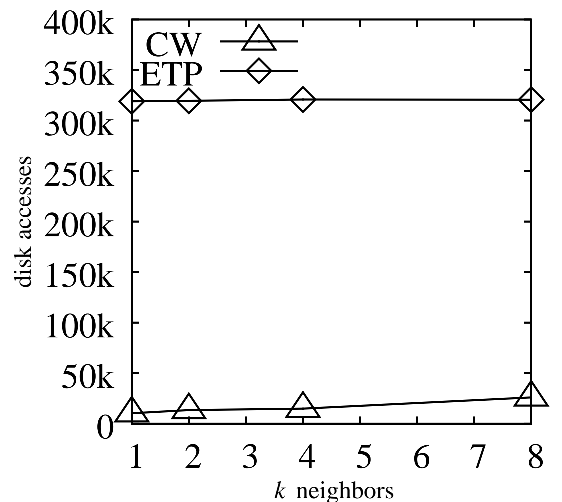


Figure 5.22: Disk accesses vs. number of neighbors for uniform data.

Figure 5.23 gives the results of an experiment to study the affect of extra neighbors on performance for the CW algorithm. Recall that the parameter x of the

Continuous_Windowing_Knn() algorithm presented in Figure 5.3 determines the size of the initial within set W (i.e., $|W| = k+x$). For a nearest neighbors query of $k = 1$, Figure 5.23 shows disk accesses on y-axis versus the number of extra neighbors x on the x-axis. The $\triangle$ symbols indicate the number of disk accesses for aircraft data, and the $\diamond$ symbols indicate disk accesses for uniform data. It appears that small increases in the number of extra neighbors reduce performance for uniform data, but have little affect on the non-uniform aircraft data.

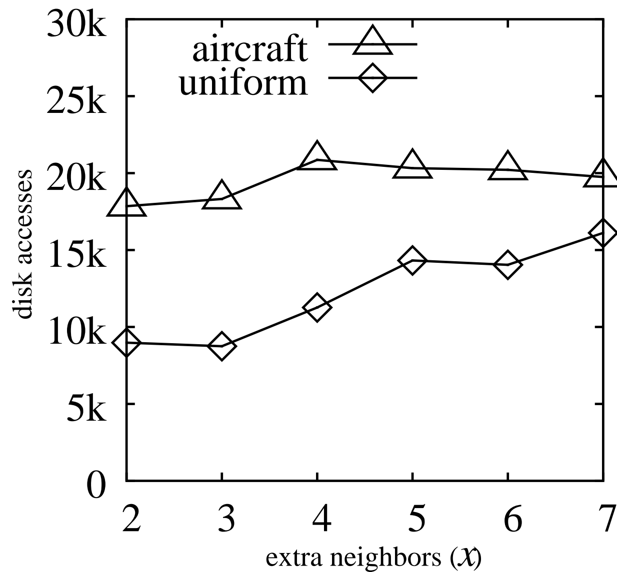


Figure 5.23: Disk accesses vs. extra neighbors for CW algorithm.

The purpose of the next experiment is to determine how the size of the disk cache affects performance for each algorithm. Figure 5.24 shows the results of varying the cache size for the event queue of the CW algorithm. Figure 5.25 shows the results of varying the cache size for the TPR-tree of the ETP algorithm. Each figure plots disk accesses on the y-axis versus the number of pages in the cache on the x-axis. The $\triangle$ symbols indicate disk accesses for aircraft data, while the $\diamond$ symbols indicate disk accesses for uniform data. This experiment was used in choosing the default

disk cache sizes for other experiments.

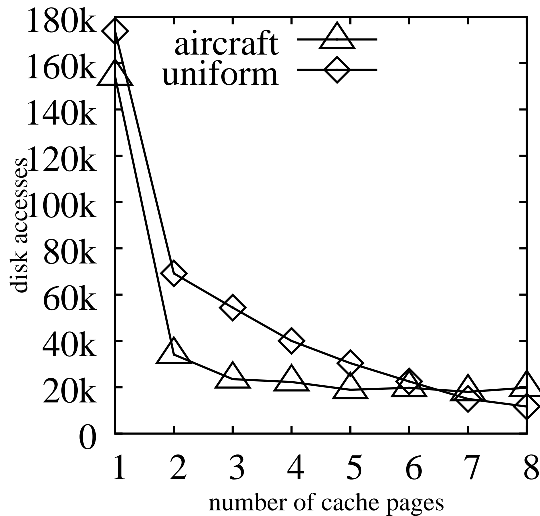


Figure 5.24: Disk accesses vs. number of disk cache pages for CW algorithm.

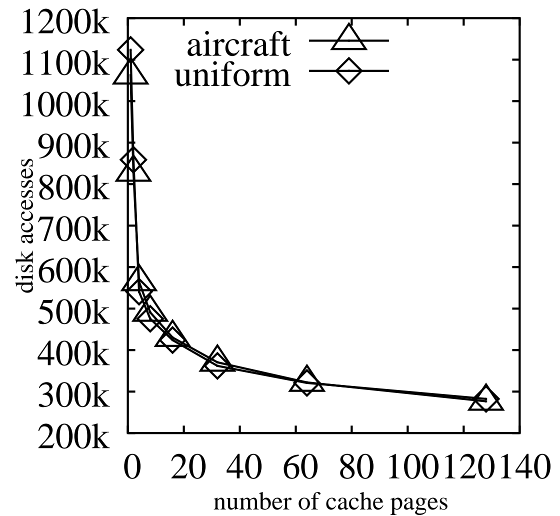


Figure 5.25: Disk accesses vs. number of disk cache pages for ETP algorithm.

The purpose of the final experiment is to study the relative performance of the to algorithms as the average update interval (UI) changes. Figure 5.26 gives the results for uniform data. This experiment was run on just synthetic data since precise control of the UI is much easier for synthetic data than for real data. In Figure 5.26, disk accesses (y-axis) are plotted against the average update interval (UI). The $\triangle$ symbols show disk accesses for the CW algorithm while $\diamond$ symbols show disk accesses for the ETP algorithm. It appears from the graph that changes in UI have very little affect on the CW algorithm, whereas the ETP algorithm appears to grow exponentially with respect to UI. This is a significant improvement for the improved CW algorithm presented here from original CW algorithm presented in [35]. In [35] the CW did not have a dynamic within query window, but rather a window of a fixed radius was used for all queries. This resulted in a within set that was much larger than it had to be so underflow could be avoided. The result in [35]

was apparent exponential growth in the number of disk accesses with respect to the average update interval UI. The ability of the improved CW algorithm presented in this section to adjust the window size dynamically, results in nearly constant performance factor for different values for UI.

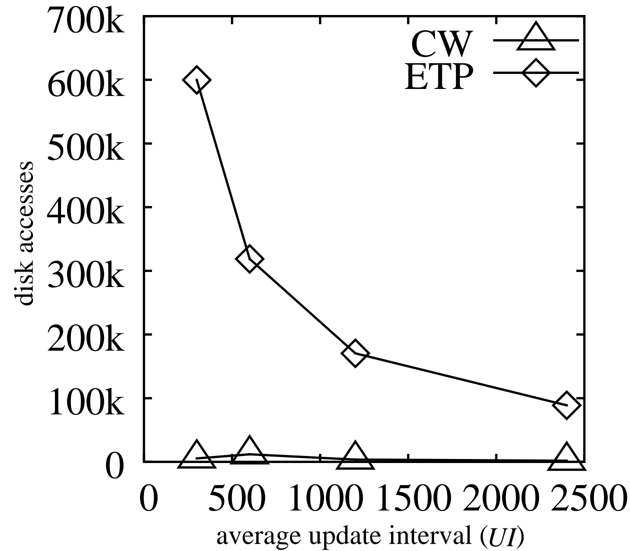


Figure 5.26: Disk accesses vs. average update interval for uniform data.

5.7 Conclusion

In this chapter we presented an improved version of the the continuous Windowing k-NN algorithm (CW). We extend the algorithm originally presented in [35] to support updates to the query point, and to dynamically adjust the size of the within set. We also extended the ETP algorithm presented in [35] to support updates toe the query point for comparison. The improved CW method outperformed the improved ETP algorithm when updates occur during query maintenance by more than an order of magnitude.

Chapter 6

Spatial Join Queries

Consider the following queries. For each airplane, keep track of every other airplane that is too close for safety. For each tank, keep track of each target that is within firing range. For each robot explorer in a swarm of robots, keep track of all neighboring robots that are within radio range. For each unmanned air vehicle, keep track of every observation target within 5 miles. Each of these example queries is a spatial join query that is maintained over time.

In Chapter 2 we introduced the concept of w-events and event-based query maintenance. In Chapter 5 we used them to support k-nn queries and introduce an event-based query maintenance algorithm (see Figure 5.3 and 5.10) to support updates. In this section we generalize the event-based query maintenance algorithms given in Figures 5.3 and 5.10 to maintain spatial join queries on moving points while the base relations are updated throughout the duration of the query. As in Chapter 5, it is assumed there is no previous knowledge about the updates prior to the

arrival of the update. We also introduce a new concept of an *event generation cycle* to help reduce the size of the event queue.

Given relations $l(L)$ and $r(R)$, a spatial join is the join $l \bowtie_{\|\alpha_l,\alpha_r\| \leq d} r$ where $\alpha_l \in L$ and $\alpha_r \in R$ are the spatial attribute names of moving points in the relational schemas, $d \geq 0$, and $\|\alpha_l, \alpha_r\|$ is the Euclidean distance metric.

Figure 6.1 is an example of a continuous spatial join on 1-dimensional moving points where $l = a, b$ and $r = x, y$ at time $ct = 2$. Each row in the figure shows the state of the moving objects as the current time ct advances. The join distance is illustrated by the shaded crosshatched areas extending for 8 distance units around points in relation r. Points b, and x are moving at speed 1. Point a is moving at speed 2. The arrows indicate their direction of movement. Although the example is 1-dimensional, the algorithms presented in this paper work for any dimension $D > 0, D \in \mathbb{N}$.

Table 6.1 shows how updates and events affect the query result as time advances. Column 1 is the current time ct. Column 2 gives the event or update at time ct. Column 3 gives the join result J after the event or update occurs. For brevity, only the moving point attributes of tuples are presented.

The solution to this problem centers on how to refresh the precomputed events as well as the query result when the spatial join base relations are updated. Simple application of previous work (e.g., [65]) is not sufficient as shown by the results of experiments in Section 6.5. To our knowledge, there is no previous work on maintaining spatial joins on continuously moving points as the relations are updated.

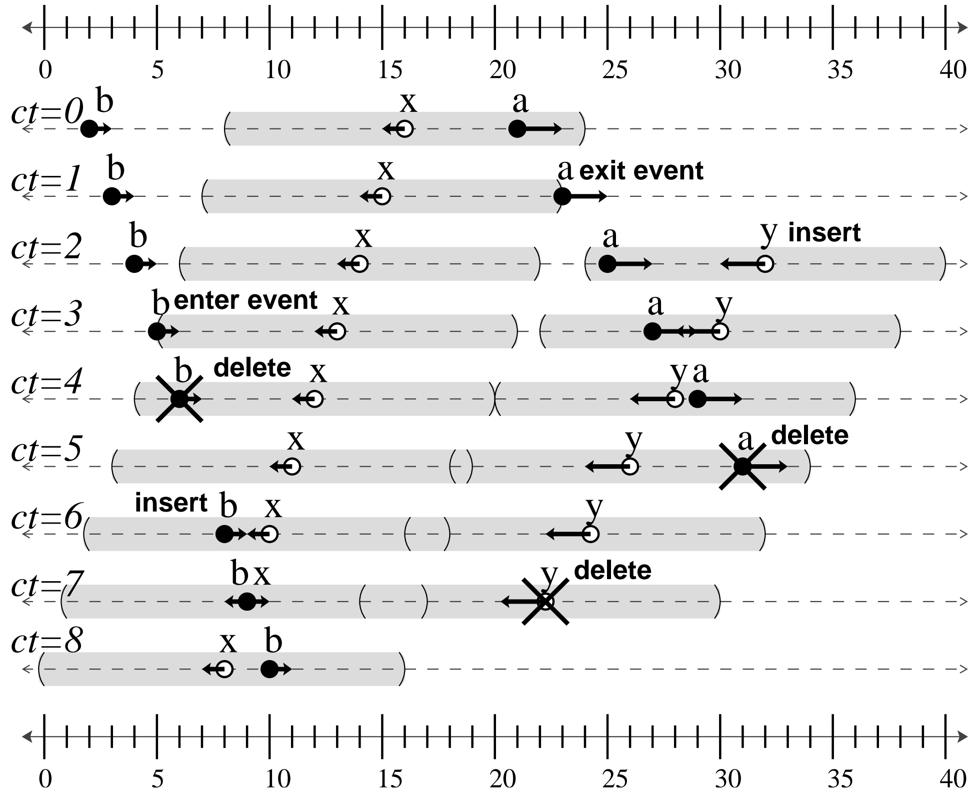


Figure 6.1: Example spatial join of 1-dimensional moving points.

time ct	event or update	J
0	initial join	$\{\langle a, x\rangle\}$
1	exit event	$\{\}$
2	insert y into r	$\{\langle a, y\rangle\}$
3	enter event	$\{\langle a, y\rangle, \langle b, x\rangle\}$
4	delete b from l	$\{\langle a, y\rangle\}$
5	delete a from l	$\{\}$
6	insert b into l	$\{\langle b, x\rangle\}$
7	delete y from r	$\{\langle b, x\rangle\}$
8	no change	$\{\langle b, x\rangle\}$

Table 6.1: Trace of the example join query given in Figure 6.1.

Recall that event-based query maintenance involves the processing of events and updates to maintain a consistent query result as time advances. For example, in the sample query in Figure 6.1, events occur at times 1 and 3 resulting in a change to the

query result. The occurrences of one or more events are computed in advance and placed in a priority queue sorted by time. Thus, when the time of the event arrives, the query result is modified, and more events are possibly computed. Updates may also change the query result. For example, in the sample query in Figure 6.1, updates occur at times 2, 4, 5, and 6 resulting in a change to the query result. The query result can be updated using techniques similar to incremental view maintenance techniques [25] when an update occurs. Additionally, the events in the queue must also be modified as a result of updates to keep them correct. Adjusting events and the spatial join query result when updates occur is the focus of this chapter. Support for updates to the base relations is what distinguishes our work from previous work on maintaining spatial joins on continuously moving points (i.e., [65]).

In Section 6.1 we present the general query engine. Two approaches based on the query engine are the presented. The first approach is an extension of the *continuous spatial join* [65] algorithm to support updates (Section 6.2). Our own novel approach is presented in Section 6.3. Performance issues are discussed in Section 6.4, and experimental results are presented in Section 6.5.

6.1 Query Engine

The *continuous spatial join with updates* query engine, given in Figure 6.2, maintains continuous event-driven spatial join queries on moving points while supporting updates.[1] The algorithm starts by computing the initial join result and events to

[1]For brevity, we do not consider simultaneous events and/or updates.

be processed. It then processes events and updates to maintain the query result as time advances. Note the similarity between the CSJU() algorithm and the top-level of the Continuous_Windowing_Knn() algorithm presented in Figure 5.3. The primary difference is that the CSJU() algorithm supports a binary query operator, whereas the Continuous_Windowing_Knn() supports a unary operator on one relation.

```
    procedure CSJU(j)
 1. Perform initial join and report result J.
 2. Generate_Events(j)
 3. while true do
 4.    Sleep until there is an update, or an event comes due,
       or the end of the event generation cycle is reached.
 5.    if an event came due then
 6.       Process_Next_Event(j)
 7.    else if there is an update then
 8.       if tuple was inserted in j.l then
 9.          Insert_L(j)
10.       else if tuple was deleted from j.l then
11.          Delete_L(j)
12.       else if tuple was inserted in j.r then
13.          Insert_R(j)
14.       else /* tuple was deleted from j.r */
15.          Delete_R(j)
16.       end if-then-else
17.    else /* end of event generation cycle was reached */
18.       Generate_Events(j)
19.    end if-then-else
20. end while
```

Figure 6.2: CSJU()

Input parameter j to the CSJU() algorithm defines the continuous spatial join query. Associated with j are the two relations $j.l$ and $j.r$ to be joined. Relation $j.l$ has a TPR-tree index $j.tpr_l$ on moving point attribute $\alpha_l \in L$ where L is the schema of $j.l$. Relation $j.r$ has a TPR-tree index $j.tpr_r$ on moving point attribute

$\alpha_r \in R$ where R is the schema of $j.r$. Events are stored in a queue $j.Q$ sorted by time in increasing order. The scalar value $j.dist$ is the join distance, and scalar $j.generation_length$ is the event generation cycle duration (described below).

When the CSJU() algorithm in Figure 6.2 is initially called, no query result has been computed yet, and thus the event queue $j.Q$ is empty. Therefore, it starts by computing the initial query result for the current time (line 1). In other words, for relations $j.l(L)$ and $j.r(R)$, the spatial join $J = (j.l \bowtie_{pred} j.r)$ is computed where $pred = (\|\alpha_l, \alpha_r\| \leq j.dist)$ is the join predicate, $\alpha_l \in L$ and $\alpha_r \in R$ are the attribute names of the moving points in their schemas, $j.dist > 0$ is the join distance, and $\|\alpha_l, \alpha_r\|$ is the Euclidean distance metric. Once the initial query result J is computed, then the events needed to maintain the query result J during the current event generation cycle are computed in a call to Generate_Events(j) (line 2). Time is divided into segments of equal length called *event generation cycles*. Only events that occur during the current event generation cycle are considered for processing. The first event generation cycle starts at the current time (e.g., now), and lasts for the next $j.generation_length$ time units. The details on what events get placed on the queue $j.Q$ by Generate_Events(j) depend on the approach used (*All Events* (AE) see Section 6.2, or *Next Event* (NE) see Section 6.3). For now, all the reader needs to know is that events will be enqueued before they occur.

The main loop of CSJU() in Figure 6.2 repeatedly processes events and updates to maintain the query result J as time advances. An update is a tuple deletion, or insertion in a relation. At the beginning of the loop, the algorithm will sleep

until either an event comes due, an update takes place, or the end of the current event generation cycle is reached. If the end of the current event generation cycle is reached, then Generate_Events(j) is invoked again to enqueue the events for the next event generation cycle.

Updates are processed according to the type of update (lines 8–16). An insertion update to relation $j.l$ is processed in a call to Insert_L(j) (line 9). Insert_L(j) maintains the consistency of the query result J with respect to the update. The details on how Insert_L(j) works depends on the approach used (AE Section 6.2, or NE Section 6.3). Similarly, Insert_R(j) (line 13) processes an insertion to relation $j.r$, Delete_L(j) (line 11) processes a deletion from relation $j.l$, and Delete_R(j) (line 15) processes a deletion from relation $j.r$. Again, these procedures maintain the correctness of the query result J with respect to updates. Details on how these procedures work can be found in following sections for each approach.

If an event comes due, then the event is processed by invoking Process_Next_Event(j) (line 6). Process_Next_Event(j) updates the query result J using the next event on $j.Q$ to maintain correctness as time advances. The details on how procedure Process_Next_Event(j) operates depends on the approach used (AE described in Section 6.2, or NE described in Section 6.3).

W-events are placed on the priority queue $j.Q$. In the context of the spatial join query $J = (j.l \bowtie_{pred} j.r)$, where join predicate $pred = (\|\alpha_l, \alpha_r\| \leq j.dist)$, a w-event is denoted $\mathsf{w}(p_l, p_r, t)$, where p_l represents an instance of moving point attribute α_l, p_r represents an instance of moving point attribute α_r, and t is the time at which p_l

and p_r move to be at the distance *j.dist* of each other. For example, the enter event in line $ct = 3$ of Figure 6.1 is denoted $\mathsf{w}(b, x, 3)$, and the exit event in line $ct = 1$ of Figure 6.1 is denoted $\mathsf{w}(a, x, 1)$.

The design of the procedures in Figure 6.2 that take j as their argument vary with the two approaches *All Events* (AE) or *Next Event* (NE) presented below in Sections 6.2, and 6.3, respectively. The basic differences between AE and NE are the times at which events are computed and placed on the queue. Both of the approaches presented below follow the general strategy of only changing events on the queue directly affected by a particular update. The difference lies in which events are put on the queue in the first place, and consequently the events on the queue are modified when an update occurs. The AE algorithm enqueues all events up to the end of the event generation cycle. The NE algorithm enqueues only the next event to occur for each moving point in one designated join relation. By convention this is the left relation *j.l*. Again, neither approach enqueues any event that occurs beyond the end of the current event generation cycle.

It is worth mentioning that updates could be supported in a simple modification to Tao and Papadias's algorithm for continuous spatial joins [65] (see Section 3.7.3) by discarding any previously computed events, and recomputing all of them when an update occurs. This alternative turns out to be much less efficient than our AE or NE approach as shown by the experiments described in Section 6.5.

6.2 All Events (AE) Approach

The *All Events* (AE) approach maintains all currently pending events on the queue that occur between the current time and the end of the current event generation cycle. This algorithm can be thought of as an extension of the continuous spatial join (CSJ) algorithm for future queries presented in [65] (see Section 3.7.3) to support updates. The approach of this extension is to run the CSJ future query for some finite time in the future to find all the events in that time period and place them on an event queue. If an update occurs, modify the query result, and modify the event queue to reflect the change introduced by the update. When updates occur, the event queue is modified to maintain consistency between it and the new state of the join relations. In this section we describe the AE versions of the procedures. The NE versions of the procedures are described in Section 6.3.

Table 6.2 shows a trace for the example from Figure 6.1 using the AE approach. For this example, assume each event generation cycle is 12 time units long. The first cycle starts at time $ct = 0$, so the end of the first generation cycle is at time 12. Column 1 is the current time ct. Column 2 gives the event or update at time ct. Column 3 gives the join result J, and column 4 shows the contents of the event queue $j.Q$ for the AE approach after each event or update is processed. Notice when y is inserted into $j.r$ at time $ct = 2$ there is no exit event inserted in the queue for point b. This is because the exit event for b and y does not occur until the next event generation cycle.

The AE version of Process_Next_Event() is given in Figure 6.3 (called from line 6 of

time ct	event or update	J	$j.Q$
0	initial join and event gen.	$\{\langle a,x\rangle\}$	$\langle \mathsf{w}(a,x,1), \mathsf{w}(b,x,3)\rangle$, $\mathsf{w}(b,x,11)\rangle$
1	exit event	$\{\}$	$\langle \mathsf{w}(b,x,3), \mathsf{w}(b,x,11)\rangle$
2	insert y into $j.r$	$\{\langle a,y\rangle\}$	$\langle \mathsf{w}(b,x,3), \mathsf{w}(a,y,5.75)$, $\mathsf{w}(b,y,8.\overline{6}), \mathsf{w}(b,x,11)\rangle$
3	enter event	$\{\langle a,y\rangle, \langle b,x\rangle\}$	$\langle \mathsf{w}(a,y,5.75), \mathsf{w}(b,y,8.\overline{6})$, $\mathsf{w}(b,x,11)\rangle$
4	delete b from $j.l$	$\{\langle a,y\rangle\}$	$\langle \mathsf{w}(a,y,5.75)\rangle$
5	delete a from $j.l$	$\{\}$	$\langle\rangle$
6	insert b into $j.l$	$\{\langle b,x\rangle\}$	$\langle \mathsf{w}(b,y,8.\overline{6}), \mathsf{w}(b,x,11)\rangle$
7	delete y from $j.r$	$\{\langle b,x\rangle\}$	$\langle \mathsf{w}(b,x,11)\rangle$

Table 6.2: Trace of the *All Events* (AE) algorithm for the example from Figure 6.1.

Figure 6.2). This handles the processing of events as they happen. The query result J is updated based on the event type. The function $\mathsf{pop}(j.Q)$ (line 1) dequeues the next w-event from $j.Q$ and returns it. If the event is an enter event, then a new tuple is added to the join result. If it is an exit event, then a tuple is removed from the join result. The function $\mathsf{pair}(e)$ (lines 3 and 5) joins the tuples that contain the moving points involved in the event e and returns them. For example, $\mathsf{pair}(\mathsf{w}(b,x,3)) = \langle b,x\rangle$. For brevity, only the moving point attributes are used to represent the whole tuple.

In Figure 6.3, once an event has been processed, no new events need to be placed on the queue since all the events for the current event generation cycle are already enqueued. This is why this approach is called the *All Events* approach, as all the events from now until the end of the current event generation cycle are already on the queue. Row $ct = 1$ in Table 6.2 is an example of an exit event. Row $ct = 3$ in Table 6.2 is an example of an enter event.

The AE version of $\mathsf{Insert_L}(j)$ (called from line 9 of Figure 6.2) is given in Figure 6.4. This is invoked when there has been an insertion to relation $j.l$. $\mathsf{Insert_L}(j)$

```
procedure AE_Process_Next_Event(j)
1.  Event e ←pop(j.Q)
2.  if event e is an enter event then
3.     Report pair(e) inserted into result J.
4.  else if event e is an exit event then
5.     Report pair(e) deleted from result J.
```

Figure 6.3: AE_Process_Next_Event(j)

performs a within query to update the join query result J, then finds all the new events involving the newly inserted point, and inserts them in the queue. The within query is performed in the call to Find_Within_Dist($j.tpr_r, j.r, p_l, j.dist$) (line 4). Find_Within_Dist() is a within distance query that returns the tuples from $j.r$ with points indexed by $j.tpr_r$ that are within distance $j.dist$ of query point p_l. These tuples are joined with the newly inserted tuple τ_l and added to the join query result J (line 5). The call to function All_Within_Events($j.tpr_r, p_l, j.dist, \Delta t$) (line 6) returns all the w-events at distance $j.dist$ from query point p_l during the next Δt with the moving points indexed by $j.tpr_r$. Procedure All_Within_Events() is implemented using the *incremental within event query* described in Section 3.7.3, stopping when the max time Δt is reached. All new events found are placed on the queue $j.Q$ (line 7). Finally, the new tuple's moving point attribute p_l is added to the index $j.tpr_l$ (line 8). Row $ct = 6$ in Table 6.2 is an example of an insertion to $j.l$.

The AE version of the Delete_L() (called from line 11 of Figure 6.2) is given in Figure 6.5. It is invoked when there has been a deletion from relation $j.l$. The algorithm removes all events involving the instance p_l of the moving point attribute α_l of tuple τ_l from the queue (line 3). It then performs a within query using p_l as

procedure AE_Insert_L(j)
1. Let τ_l be the new tuple that was inserted into $j.l$.
2. Let p_l be the instance of moving point attribute α_l in τ_l.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. **foreach** tuple $\tau \in$ Find_Within_Dist($j.tpr_r, j.r, p_l, j.dist$)
5.   Report joined tuple $\tau_l\tau$ inserted into result J.
6. **foreach** event $e \in$ All_Within_Events($j.tpr_r, p_l, j.dist, \Delta t$)
7.   Enqueue event e on queue $j.Q$.
8. Insert point p_l into index $j.tpr_l$.

Figure 6.4: AE_Insert_L(j)

the query point in a call to Find_Within_Dist() (line 4). The join of the newly deleted tuple τ_l and the tuples found in the within query are removed from the join query result J (line 5). Finally, p_l is removed from index $j.tpr_l$ (line 6). Rows $ct = 4$ and $ct = 5$ of Table 6.2 are examples of deletions from relation $j.l$.

procedure AE_Delete_L(j)
1. Let τ_l be the tuple that was deleted from $j.l$.
2. Let p_l be the instance of moving point attribute α_l in τ_l.
3. Remove all events involving p_l from $j.Q$.
4. **foreach** tuple $\tau \in$ Find_Within_Dist($j.tpr_r, j.r, p_l, j.dist$)
5.   Report joined tuple $\tau_l\tau$ deleted from result J.
6. Delete point p_l from index $j.tpr_l$.

Figure 6.5: AE_Delete_L()

The AE version of Insert_R() (called from line 13 of Figure 6.2) is symmetric with AE_Insert_L(j). In particular, replacing $\tau_l\tau$ with $\tau\tau_r$ in line 5 of Figure 6.4, followed by swapping symbols l with r in Figure 6.4 yields the AE_Insert_R(j) algorithm. Row $ct = 2$ of Table 6.2 is an example of an insertion update to $j.r$. Similarly, the AE version of Delete_R() (called from line 15 of Figure 6.2) is symmetric with procedure

AE_Delete_L(j). In particular, AE_Delete_R(j) is derived from AE_Delete_L(j) by replacing $\tau_l\tau$ with $\tau\tau_r$ in line 5 of Figure 6.5, and then swapping symbols l with r in Figure 6.5. Row $ct = 7$ of Table 6.2 is an example of a deletion update to $j.r$.

The AE algorithm for Generate_Events(j) (called from line 18 of Figure 6.2) is given in Figure 6.6. It is invoked at the beginning of each event generation cycle to populate the event queue $j.Q$ with new events. Note that $j.Q$ is necessarily always empty when Generate_Events(j) is invoked. All events occurring during the new event generation cycle are found and enqueued on the event queue $j.Q$. This is done by scanning all the leaf nodes of index $j.tpr_l$ (line 2). Each point p_l found in a leaf node (line 3) is used as the query point to All_Within_Events() (line 4). The resulting events are placed on the event queue $j.Q$ (line 5).

procedure AE_Generate_Events(j)
1. Let Δt be the time between now and the end of the current event generation cycle.
2. **foreach** leaf node $n \in j.tpr_l$
3.   **foreach** moving point $p_l \in n$
4.   **foreach** event $e \in$ All_Within_Events($j.tpr_r, p_l, j.dist, \Delta t$)
5.   Enqueue event e in queue $j.Q$.

Figure 6.6: AE_Generate_Events(j)

6.3 Next Event (NE) Approach

The *Next Event* (NE) approach enqueues just the next pending event during the current event generation cycle for each moving point from one designated join relation, relation $j.l$ by convention. In other words, there is at most one event, the next event,

for each moving point in relation $j.l$. Table 6.3 shows a trace of the NE approach with respect to the example given in Figure 6.1. This example assumes each event generation cycle is 12 time units long. The first cycle starts at time $ct = 0$, so the end of the first cycle is at time 12. Column 1 is the current time ct. Column 2 gives the event or update at time ct. Column 3 gives the join result J, and column 4 shows the contents of $j.Q$ for the NE approach after each event or update is processed. As with the AE approach, the event queue is modified after an update to maintain consistency between it and the new state of the join relations.

time ct	event or update	J	$j.Q$
0	initial join and event gen.	$\{\langle a,x\rangle\}$	$\langle \mathsf{w}(a,x,1),\ \mathsf{w}(b,x,3)\rangle$
1	exit event	$\{\}$	$\langle \mathsf{w}(b,x,3)\rangle$
2	insert y into $j.r$	$\{\langle a,y\rangle\}$	$\langle \mathsf{w}(b,x,3),\ \mathsf{w}(a,y,5.75)\rangle$
3	enter event	$\{\langle a,y\rangle,\langle b,x\rangle\}$	$\langle \mathsf{w}(a,y,5.75),\ \mathsf{w}(b,y,8.\overline{6})\rangle$
4	delete b from $j.l$	$\{\langle a,y\rangle\}$	$\langle \mathsf{w}(a,y,5.75)\rangle$
5	delete a from $j.l$	$\{\}$	$\langle\rangle$
6	insert b into $j.l$	$\{\langle b,x\rangle\}$	$\langle \mathsf{w}(b,y,8.\overline{6})\rangle$
7	delete y from $j.r$	$\{\langle b,x\rangle\}$	$\langle \mathsf{w}(b,x,11)\rangle$

Table 6.3: Trace of the *Next Event* (NE) algorithm for the example from Figure 6.1.

Figure 6.7 gives the NE version of Process_Next_Event() (called from line 6 of Figure 6.2). This handles the processing of events as they occur. The query result J is updated based on the event type of e. Then, the next event for the moving point from relation $j.l$ involved in e is enqueued, if one exists. The function pop($j.Q$) in line 1 dequeues the next w-event from $j.Q$ and returns it. If the event is an enter event, then a new tuple is added to the join result (line 5). If the event is an exit event, then a tuple is removed from the join result (line 7). The function pair(e) joins the tuples that contain the moving points involved in event e and returns them.

procedure NE_Process_Next_Event(j)
1. Event $e \leftarrow$pop($j.Q$)
2. Let p_l be the moving point in $j.l$ involved in e.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. **if** event e is an exit event **then**
5. Report pair(e) deleted from result J.
6. **else if** event e is an enter event **then**
7. Report pair(e) inserted into result J.
8. Let e_{exit} be the exit event following e for the points involved in e.
9. **if** event e_{exit} occurs before now + Δt **then**
10. Let Δt be the time between now and e_{exit}.
11. **end if-then-else**
12. $e_{next} \leftarrow$ Next_Within_Event($j.tpr_r, p_l, j.dist, \Delta t$)
13. **if** event e_{next} is not null **then** enqueue e_{next} in $j.Q$.

Figure 6.7: NE_Process_Next_Event(j)

In the case of an enter event, an optimization is made in NE_Process_Next_Event() in finding the next event e_{next}. Observe that every enter event between moving points is followed by an exit event sometime in the future. An exit event e_{exit} for the points involved in e can be computed using the information already in memory. The time of e_{exit} is an upper bound on the time of the next event e_{next}. Event e_{exit} is used to shorten Δt if it occurs before the end of the current event generation cycle (line 10). The call to Next_Within_Event($j.tpr_r, p_l, j.dist, \Delta t$) (line 12) returns the next w-event at distance $j.dist$ from query point p_l during the next Δt time units with the moving points indexed by $j.tpr_r$. It returns a null event if no such next event exists. Procedure Next_Within_Event() is implemented using the *incremental within event query* described in Section 3.7.3, stopping after the first event is returned. Row $ct = 1$ in Table 6.3 is an example of an exit event. Row $ct = 3$ in Table 6.3 is an

example of an enter event.

Figure 6.8 gives the NE version of Insert_L() (called from line 9 of Figure 6.2). It is invoked when there has been an insertion to relation $j.l$. First, the instance p_l of the moving point attribute α_l in the inserted tuple τ_l is used as the query point in a call to Find_Within_Dist() to perform a within query (line 4). The result of the within query is used to update the join query result J (line 5). Then the next event for p_l is found by calling Next_Within_Event() (line 10). If null is returned by Next_Within_Event(), then no such next event for p_l exists, otherwise the non-null event is enqueued in $j.Q$ (line 11). Finally, p_l is added to the index $j.tpr_l$ (line 12).

procedure NE_Insert_L(j)
1. Let τ_l be the new tuple that was inserted into $j.l$.
2. Let p_l be the instance of the moving point in τ_l.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. **foreach** $\tau \in$ Find_Within_Dist($j.tpr_r, j.r, p_l, j.dist$)
5. Report joined tuple $\tau_l\tau$ inserted into result J.
6. Let e_{exit} be the exit event between the points in τ_l and τ.
7. **if** event e_{exit} occurs before now + Δt **then**
8. Let Δt be the time between now and e_{exit}.
9. **end foreach**
10. $e_{next} \leftarrow$Next_Within_Event($j.tpr_r, p_l, j.dist, \Delta t$)
11. **if** event e_{next} is not null **then** enqueue e_{next} in $j.Q$.
12. Insert point p_l into index $j.tpr_l$.

Figure 6.8: NE_Insert_L(j)

In the case when the new moving point is joined with a moving point in $j.r$, an optimization can be performed to find the next event e_{next}. As in the optimization for NE_Process_Next_Event() described above, an exit event for each join pair can be computed using the information already in memory. The time of the earliest exit

event is an upper bound on the time of the next event e_{next}. If it occurs before the end of the current event generation cycle, it is used to shorten Δt (line 8). Row $ct = 6$ of Table 6.3 is an example of an insertion update to relation $j.l$.

Procedure NE_Delete_L(j) is the NE version of procedure Delete_L() (called from line 11 of Figure 6.2). It is called when there has been a deletion from relation $j.l$. It is identical to AE_Delete_L(j) given in Figure 6.5.

Figure 6.9 gives the NE version of Insert_R() (called from line 13 of Figure 6.2). It is invoked when there has been an insertion to relation $j.r$. First, the instance p_r of the moving point attribute α_r in the newly inserted tuple τ_r is used as the query point in a call to Find_Within_Dist() to perform a within query (line 4). The result of the within query is used to update the join query result J (line 5). Then all events involving the newly inserted point p_r and points in $j.l$ are checked to see if any of them take place before the currently queued events for points in $j.l$. To do this, the function All_Within_Events() (line 6) is called to find the events. All the events for p_r must be examined to determine if any of them will be added to the queue. If a new event e involves a point p_l from relation $j.l$, and p_l is not involved in any other event already on queue $j.Q$, then e is added to the queue $j.Q$ (line 9). On the other hand, if the point p_l is involved in another event on the queue e_{prev}, then the time of e_{prev} is compared to the time of e. If e occurs before e_{prev} then e_{prev} is replaced on queue $j.Q$ with event e (line 13). Row $ct = 2$ of Table 6.3 is an example of an insertion update to $j.r$.

Procedure NE_Delete_R(j) is the NE version of Delete_R() (called from line 15 of

procedure NE_Insert_R(j)

1. Let τ_r be the new tuple that was inserted into $j.r$.
2. Let p_r be the instance of moving point attribute α_r in τ_r.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. **foreach** $\tau \in$ Find_Within_Dist($j.tpr_l, j.l, p_r, j.dist$)
5.   Report joined tuple $\tau\tau_r$ inserted into result J.
6. **foreach** $e \in$ All_Within_Events($j.tpr_l, p_r, j.dist, \Delta t$)
7.   Let p_l be the point from $j.l$ that is involved in e.
8.   **if** there is no event already in $j.Q$ involving p_l **then**
9.   Enqueue event e in $j.Q$.
10.   **else**
11.   Let e_{prev} be the event involving p_l in $j.Q$.
12.   **if** e_{prev} occurs after e **then**
13.   Replace e_{prev} with e in $j.Q$.
14.   **end if**
15.   **end if-then-else**
16. **end foreach**
17. Insert point p_r into index $j.tpr_r$.

Figure 6.9: NE_Insert_R(j)

Figure 6.2). It is called when there has been a deletion from relation $j.r$. Shown in Figure 6.10, it is similar to AE_Delete_R(j) described above in Section 6.2, except that in addition to removing all the events involving p_r from $j.Q$ (line 6), it then computes the next event for the other points p_l involved in those events (line 7), and enqueues them (line 8). Note that in order to correctly compute the next event using the tpr index, the point p_r must be removed from the tpr index (line 4) before Next_Within_Event() is called in line 7.

Figure 6.11 gives the NE algorithm for Generate_Events(j) (called from line 18 of Figure 6.2). It is invoked at the beginning of each event generation cycle to populate the queue $j.Q$ with events. This algorithm is identical to AE_Generate_Events(j)

procedure NE_Delete_R(j)

1. Let τ_r be the tuple that was deleted from $j.r$.
2. Let p_r be the instance of moving point attribute α_r in τ_r.
3. Let Δt be the time between now and the end of the current event generation cycle.
4. Delete point p_r from index $j.tpr_r$.
5. **foreach** event w(p_l, p_r, t) $\in j.Q$
6. Remove w(p_l, p_r, t) from $j.Q$.
7. $e \leftarrow$Next_Within_Event($j.tpr_r, p_l, j.dist, \Delta t$)
8. Enqueue e in $j.Q$.
9. **end foreach**
10. **foreach** $\tau \in$ Find_Within_Dist($j.tpr_l, j.l, p_r, j.dist$)
11. Report joined tuple $\tau\tau_r$ deleted from result J.

Figure 6.10: NE_Delete_R()

(Figure 6.6) except for line 4. In line 4, Next_Within_Event() is called instead of All_Within_Events() (Next_Within_Event() and All_Within_Events() are described above). As a result, one event, the next w-event, involving each point p_l in $j.l$ is enqueued. This is analogous to a spatial semi-join operation [29] where each object in one relation joins with at most one object in the other relation. Note that $j.Q$ is necessarily always empty when NE_Generate_Events(j) is called.

procedure NE_Generate_Events(j)

1. Let Δt be the time between now and the end of the current event generation cycle.
2. **foreach** leaf node $n \in j.tpr_l$
3. **foreach** moving point $p_l \in n$
4. **foreach** event $e \in$ Next_Within_Event($j.tpr_r, p_l, j.dist, \Delta t$)
5. Enqueue event e in queue $j.Q$.

Figure 6.11: NE_Generate_Events(j)

6.4 Performance Issues

The asymptotic sizes of the event queues differ for the AE and NE approaches. For simplicity, assume join relations of equal size each containing n tuples. The size of the TPR-tree indexes are the same for both the AE and NE approaches, but the event queues are of different sizes. The worst case for the AE algorithm arises when the join result J starts out empty, and every pair of moving points in the Cartesian product of the join relations will be in the result set and then leave the results set at some time in the future before the next event generation cycle. This leads to two events for each pair, or $\mathcal{O}(n^2)$ events in the queue. For the NE algorithm, only the next event is computed for each point in one join relation. This means that there can be at most $\mathcal{O}(n)$ events in the queue in the worst case. This may make updates more costly for the AE approach because of a larger event queue.

Although the NE algorithm queue is smaller, there is a trade off between the size of the queue and the frequency of use of the TPR-tree indexes. When an event is processed, the AE_Process_Next_Event() procedure does not access either TPR-tree index. It only accesses the event queue to get the next event. On the other hand, NE_Process_Next_Event() not only pops the next event off the queue, but also accesses the TPR-tree index $j.tpr_r$ to find the next event so that it can be inserted in the queue.

Accessing the event queue for NE is cheaper than for AE, but NE has the additional overhead lost in use of the TPR-tree to find the next event. Given this trade off between the size of the event queue, and the frequency of use of the TPR-tree

indexes, it is not apparent which approach is better through analysis alone.

6.5 Experimental Results

In our join query experiments, we used the same data set sources as those described in Section 5.5. Table 6.4 shows summary statistics for each aircraft data set used in this chapter after conversion from samples to linear functions. Each column corresponds to a different aircraft data set. Each row is a statistic on the data sets. Row 1 is the mean number flights at any given time (μ). Row 2 is the standard deviation in the number flights (σ). Row 3 is the average update interval (UI) for each data set.

μ	1097	2212	3334	4453	9021	12690	17106
σ	81.9	165.4	148.3	330.8	680.8	962.4	1293
UI	683.3	699.4	699.3	700.7	712.8	725.1	734.6

Table 6.4: Each column corresponds to a different aircraft data set. Each row is a statistic on the data sets. Row 1 is the mean number of flights at any given time (μ). Row 2 is the standard deviation in the number of flights (σ). Row 3 is the average update interval (UI) in seconds.

6.5.1 Implementation

The priority queue of events (e.g., $j.Q$) was implemented using a disk based priority search tree. Two B+-trees are used to support deletion of events based on ids of objects involved in the event. One B+-tree is sorted by ids (id_l) of objects from the first join relation $j.l$, while the second B+-tree is sorted by ids (id_r) of objects from the second join relation $j.r$. This results in a mapping of the form $id_l \rightarrow \{id_r, t\}$, and $id_r \rightarrow \{id_l, t\}$. In addition to propagating the minimum and maximum id keys

up the internal nodes of the B+-tree, the minimum event time is also propagated up the tree. The next event in time on the priority queue is found by following the minimum event time down to the leaf containing it. The TPR-tree was implemented using code provided by Saltenis et. al. [58]. Both the TPR-tree and the B+-trees were implemented using the generalized search tree (GiST) [28] version 0.9beta1 code. The code was compiled using gcc 2.95. The experiments were run on several VLSI 80686 CPU based machines running Linux.

6.5.2 Results

The experiments measure the total number of disk accesses over the duration of a query. Since we are concerned with the maintenance portion of the query, disk accesses to compute the initial join result are not included. To make full use of the real data sets available, each data set is divided into subsets at evenly spaced time intervals. Each query was performed on combinations of these subsets, including self joins. Pairs of subsets were chosen randomly without replacement from all possible combinations for a total of 110 joins per query. Only subsets taken from the same original data set are joined in a query, so the join relations are approximately the same size for each query. The number of disk accesses was averaged to yield the experiment results for a given query. This technique was used on both the synthetic and real data sets for comparability.

Independent variables are mean data set size (μ), join distance (d), event generation cycle length (EG), query duration (t_Q), and disk cache size ($|cache|$). A disk

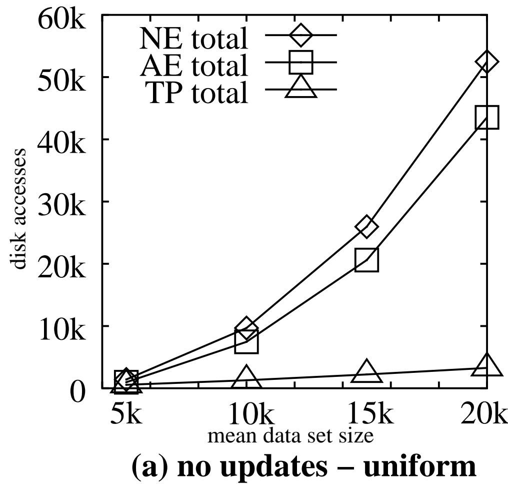
page size of 1024 bytes was used in all experiments. For aircraft flight data, the defaults are $\mu = 9021$ flights, $d = 0.08$ degrees, $t_Q = 100$ seconds, and $|cache| = 32$ pages. For synthetic uniform data, the defaults are $\mu = 10000$ points, $d = 8$ distance units, $t_Q = 100$ time units, and $|cache| = 32$ pages. Values for EG, and values other than the defaults are stated for each individual experiment below.

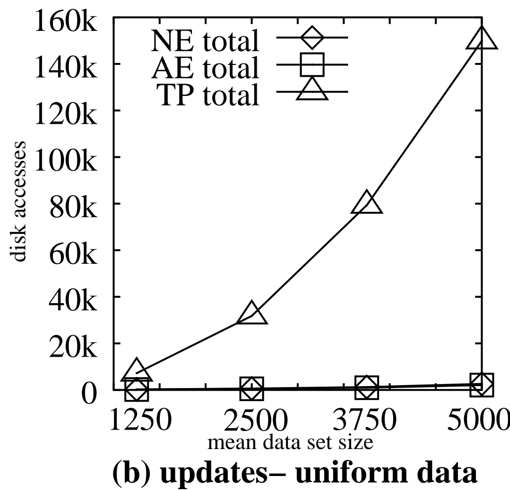
Each TPR-tree index and each B+-tree has a cache of size $|cache|$. Every cache uses a least recently used (LRU) replacement policy. For a page size of 1024 bytes, leaf nodes of the TPR-tree (B+-tree) hold 50 (22) entries, and internal nodes hold up to 28 (66) entries.

The purpose of the first experiment is to establish a baseline for performance between a simple adaptation of the state-of-the-art and the NE and AE approaches. The state-of-the-art approach is the continuous spatial join (CSJ) algorithm presented in [65]. Recall that CSJ is a future query and does not support updates (see Section 3.7.3). To support updates, the TP portion (the part that finds the events) of Tao and Papadias's CSJ algorithm is reinvoked at the time of the update to find the events for the remainder of the event generation cycle. Non-default parameters for this experiment are EG = 11, $t_Q = 10$, and $|cache| = 12$. Figure 6.12 examines the total number of disk accesses (y-axis) vs. mean number of objects at any given time (x-axis). Figure 6.12a shows the number of disk accesses in the absence of any updates to the dataset as a baseline. This is only shown for the synthetic uniform data distribution since the aircraft flight data has updates. Figure 6.12b shows the number of disk accesses for updates on the synthetic uniform data. This is only

shown for a max mean data set size of only 5000 objects since our simple adaptation of Tao and Papadias's CSJ algorithm is so expensive. Figure 6.12c shows disk accesses for updates on the aircraft flight data. As can be seen in Figure 6.12a, when there are no updates, the NE and AE approaches are about an order of magnitude less efficient than Tao and Papadias's CSJ algorithm (TP). On the other hand, when the data set is updated, Figures 6.12b and 6.12c show an advantage for the NE and AE approaches over our simple adaptation of Tao and Papadias's CSJ algorithm to support updates by nearly two orders of magnitude.

The remainder of our experiments are designed to compare the AE and NE approaches with each other. The purpose of the second experiment is to determine what influence the event generation cycle length (EG) has on the relative performance between AE and NE. This is accomplished by varying the EG using the values 1, 6, 12, 25, 50, 100, and 200 and comparing the results. All other parameters were set to the default. Figure 6.13 shows the results for aircraft flight data, while Figure 6.14 shows the results for uniform data. Graph (a) in each figure show the total number of disk accesses. This reveals that the optimal performance for both methods is around EG = 6. With a small EG value, the AE approach is better. With a large EG value, the NE approach is better. To find out why, a closer look at each disk data structure is needed. Recall that each relation in the join makes use of a TPR-tree to index the moving points. Graph (b) in each figure shows the number of disk accesses for the two TPR-tree indexes corresponding to the $j.l$ and $j.r$ relations. Recall also that the priority queue is implemented using two B+-trees. Graph (c) shows the average





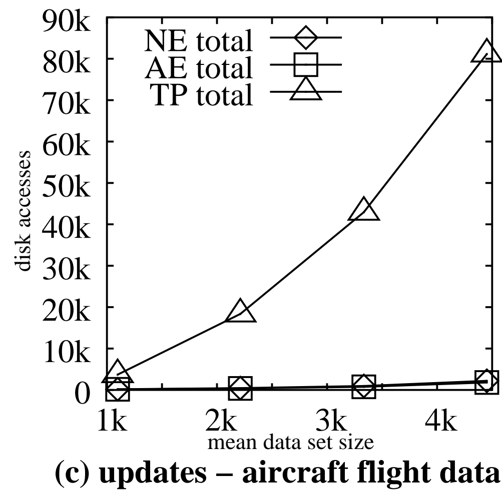
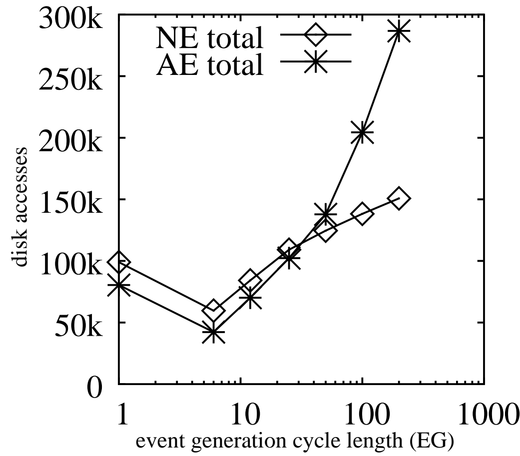
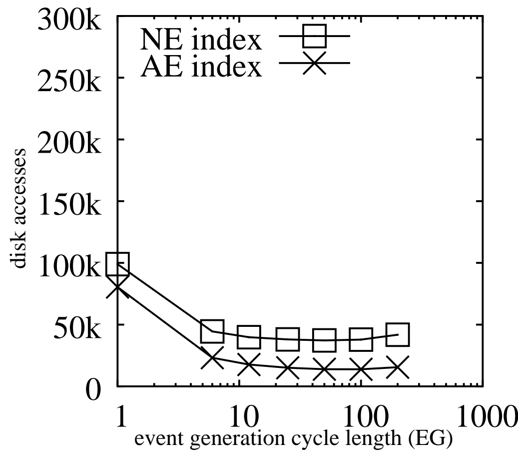


Figure 6.12: Comparison of total disk accesses for our simple adaptation of Tao and Papadias's CSJ algorithm to support updates (TP) to NE and AE without updates (a) and with updates (b and c).

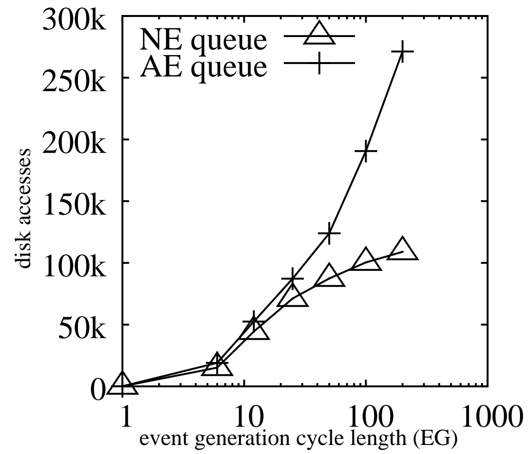
number of disk accesses of the two B+-trees for the priority event queue. For small EG values, the TPR-tree performance dominates. For a large EG values, the queue performance becomes dominant. Since the AE approach uses the event queue more heavily, and a shorter EG results in a small queue size, a small EG favors the AE approach. The NE approach uses the queue more efficiently, but accesses the TPR-tree indexes more often than the AE approach, so a large EG value favors the NE approach. The best performance is EG = 6 out of all values of EG tested.



(a) total



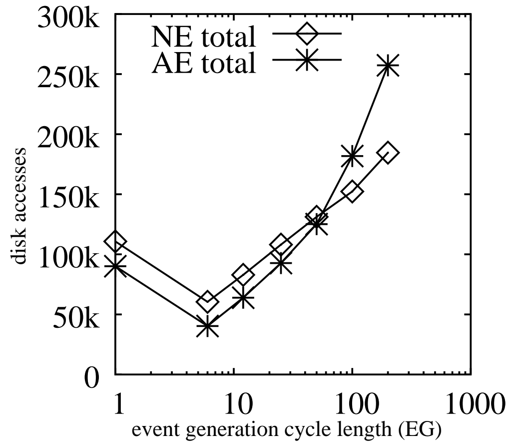
(b) index



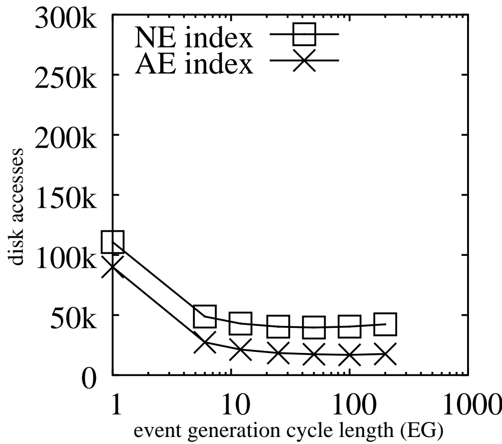
(c) queue

Figure 6.13: Aircraft flight data (x-axis is log scale)

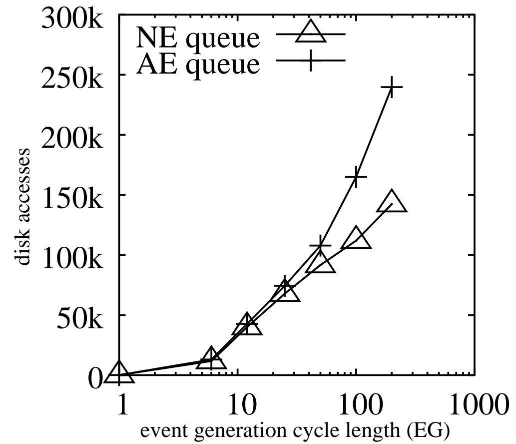
The purpose of the third experiment is to determine how query distance affects the performance of the AE and NE approaches. This is done by varying the query distance for two different EG values as shown in Figure 6.15. All other parameters were set to their default values. The uniform number of disk accesses seems to vary quadratically with distance for small values of EG, but varies linearly with distance for larger values of EG. The complex interaction between the TPR-tree indexes, and the event queue makes it unclear why linear behavior is observed for larger EG



(a) total



(b) index



(c) queue

Figure 6.14: Uniform synthetic data

values. For a small EG value, the AE approach is better (Figures 6.15a and 6.15b). For a large EG, value the NE approach is better (Figures 6.15c and 6.15d).

The purpose of the fourth and final experiment is to determine how the size of the join relations affect the performance of the AE and NE approaches. This is done by varying the data set size for two different EG values as shown in Figure 6.16. All other parameters were set to their default values. The number of disk accesses seems to follow a quadratic growth rate as a function of data set size. This is consistent with our expectations since the selectivity of a static join exhibits a quadratic growth

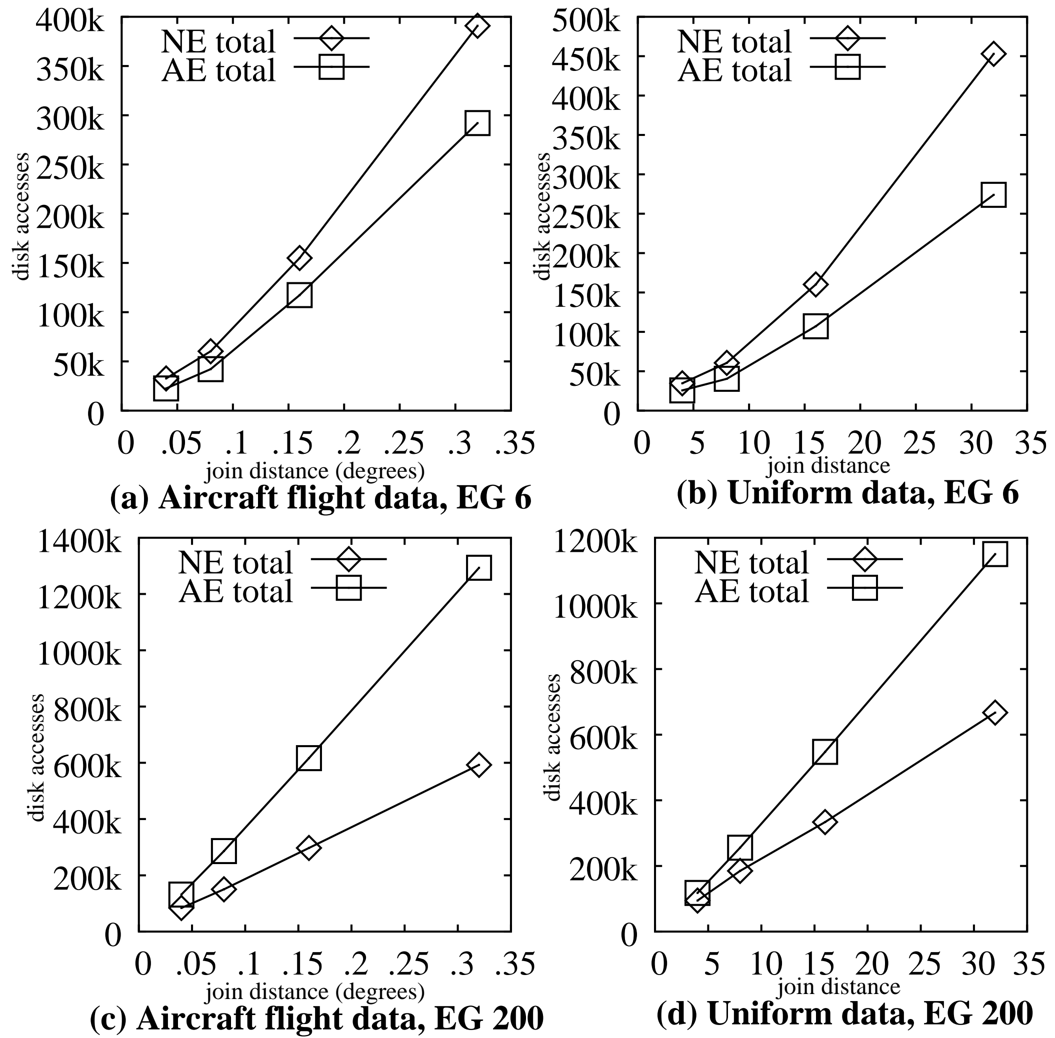


Figure 6.15: Number of disk accesses vs. join distance

rate with respect to join relation size. For a small EG value, the AE approach is better (Figures 6.16a and 6.16b). For a large EG value, the NE approach is better (Figures 6.16c and 6.16d).

6.6 Conclusion

In this chapter we extended the general event base-query processing approached from Chapter 5 to support binary queries, and we scaled the within query of the CW

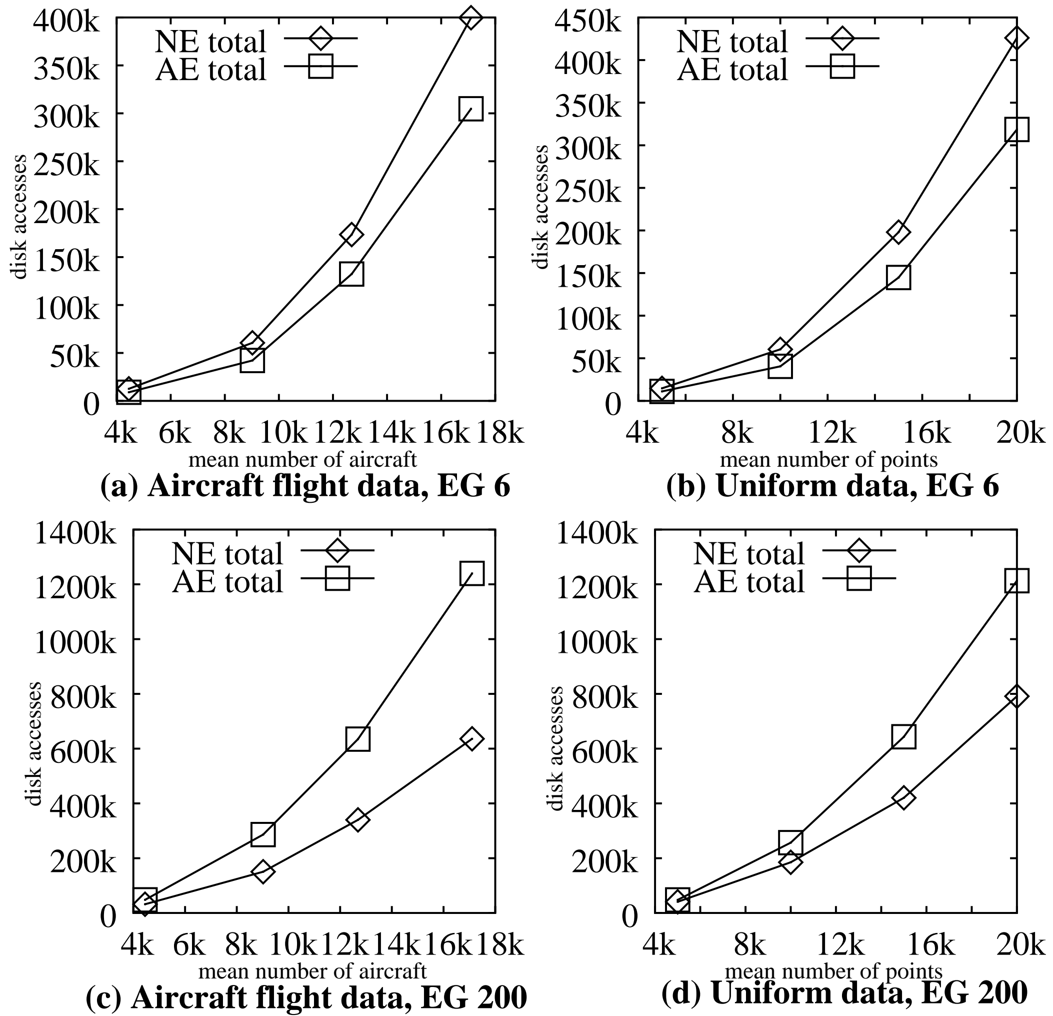


Figure 6.16: Number of disk accesses vs. mean number of moving points (per relation)

algorithm to support spatial joins. The *All Events* (AE) approach can be thought of as an adaptation of the continuous spatial join algorithm [65] to support updates in the spirit of the ETP algorithm from Chapter 5. This and our new approach, the *Next Event* (NE) approach, were shown to be nearly two orders of magnitude better than a more naive adaptation of the continuous spatial join algorithm from [65]. The relative performance of the AE and NE approaches varies depending on the tuning of an internal system parameter, namely the event generation interval length (EG). For small EG values, the AE approach is better, while for large EG values, the NE

approach is better. The AE approach performs slightly better than the NE approach in the best case (EG = 6).

Chapter 7

Spatial Semijoin Queries

7.1 Introduction

We consider the following queries. For each moving firetruck, keep track of the nearest mobile police unit. For each airplane, keep track of the nearest airport. For each cell phone, keep track of the nearest airborne relay station. For each tank, keep track of the nearest target. For each robot explorer in a swarm of robots, keep track of the nearest maintenance robot. For each unmanned air vehicle, keep track of the nearest observation objective. For each ship, keep track of the nearest sonar tracking station. These are all examples of spatial semijoin queries on moving objects. Many are examples where all the objects are moving simultaneously and continuously. All must update the query result as the objects move in real time. None know how the object will move ahead of time.

In this chapter, we address the maintenance of spatial semijoin queries over continuously moving points. Given two sets of moving points Q and D, we define

semijoin $Q \ltimes_k D$ as all the pairs $\langle q, p \rangle$, $q \in Q \wedge p \in D$, that are in Cartesian product $Q \times D$, and p is one of the k nearest neighbors of q. Set Q is the set of query points, and D is the set of data points. This amounts to a massive scaling of a continuous nearest neighbor query for all query points. Traditionally, a semijoin returns tuples from only one join relation. However, we relax this constraint to make the result meaningful in light of the examples above.

Data sets Q, and D are updated through insertions and deletions to the sets. There is no prior knowledge of what the updates will be in advance of each update occurrence. This is analogous to the maintenance of a materialized view [25], with the difference being that the query result may change as a result of the motion of points represented in the database as well as updates to the database.

Points are modeled as linear functions of time, as opposed to samples of an objects location that are updated as an object moves. Therefore, as time advances, the query result may change independently of updates. To our knowledge there has been no previous work to perform continuous spatial semijoin queries on moving objects so that any of the examples queries given in the first paragraph above for data sets of significant size can be answered. Some work on scaling k-nn queries on point data represented as samples (e.g., [52]) has been done, but not on the scale needed to perform semijoins.

In this paper, we present a new approach, termed *continuous fuzzy sets* (CFS), to perform spatial semijoins. This approach is most similar to a continuous window k-nn algorithm presented in [35]. However, CFS is not just a simple scaling of this

previous work. As we will show, previous work (e.g., [35, 65]) does not scale well. CFS is compared experimentally to a simple scaling of the time-parameterized k-nn algorithm presented in [65]. The result is a significant better performance of CFS compared to this previous work by up to an order of magnitude in some cases.

The *continuous fuzzy set* (CFS) semijoin algorithm maintains a semijoin query result $Q \ltimes_k D$ on the sets of kinematic points Q and D as time advances and updates occur. The main algorithm is a simple event-driven query processing algorithm that supports updates similar to the one presented in [35]. Events are placed on a priority queue sorted by time and dequeued one at a time for processing. There is one and only one nn-event or underflow event (described below) on the event queue for each query point in Q. Updates (insertions and deletions) are also processed as they occur. The assumption on updates is that there is no priori knowledge of updates, such as is the case in a real-time system.

The *fuzzy set* of a query point $q \in Q$ consists of all the points $S = \{s_1 ... s_n\}$, where $s_i \in D$, that are now or will be within some given distance of q sometime in the near future. This maintains a cloud of points around each query point. The next nn-event for any given point $q \in Q$ is computed from q's fuzzy set.

A fuzzy set is determined by a circle (or hypersphere for higher dimensional data) centered at q and with radius r known as the *query circle*. Radius r is chosen so that there are at least k points within Euclidean distance r of q. The points in the circle, along with points that will enter the circle sometime in the near future, make up the fuzzy set of q. Scalar value r is generally a different value for each query point

$q \in Q$. This region around the query point is denoted $\mathsf{circle}(q, r)$.

Time is divided up into uniform segments of time called fuzzy-set-intervals. The *fuzzy-set-interval* determines which points entering $\mathsf{circle}(q, r)$ belong to the fuzzy set. Only points that enter $\mathsf{circle}(q, r)$ during the current fuzzy-set-interval are in q's fuzzy set. At the start of each new fuzzy-set-interval, each query point's fuzzy set is updated (see Update_Fuzzy_Set() below).

Figure 7.1 illustrates an example fuzzy set for a single query point q in set Q, and data points {**a,b,c,d,e,f,g,h,i**} $\in D$. Assume for this example, that the length of the arrows in the figure indicate how far each point will travel in one minute. In this example, the query point is not moving for simplicity. Also, assume for this example, that the current fuzzy-set-interval will end in one minute. In this example, all the points in $\mathsf{circle}(\mathbf{q}, r)$, and all the points that will enter $\mathsf{circle}(\mathbf{q}, r)$ within the next minute (up to the end of the current fuzzy-set-interval), are in the fuzzy set of point **q**. The length of each fuzzy-set-interval is a system parameter. Points {**a,b,c,f,g,i**} are in the fuzzy set of query point **q**. Note that point **d** is closer to the circle than point **c**, but it is moving slower and will not enter the circle during the next minute.

An *underflow event* occurs when the k^{th}neighbor of some query point q leaves $\mathsf{circle}(q, r)$. When this happens, r has to be increased to encompass more data points in $\mathsf{circle}(q, r)$. Underflow events are denoted by $\mathsf{uf}(q, p_k, t)$, where $q \in Q$ is the query point, $p_k \in D$ is the current k^{th}neighbor of q, and t is the underflow event time. For example, suppose that the query for Figure 7.1 is $Q \ltimes_3 D$, that is, for each point in Q find the 3 nearest neighbors in D. Point **g** is currently the 3rd nearest neighbor from

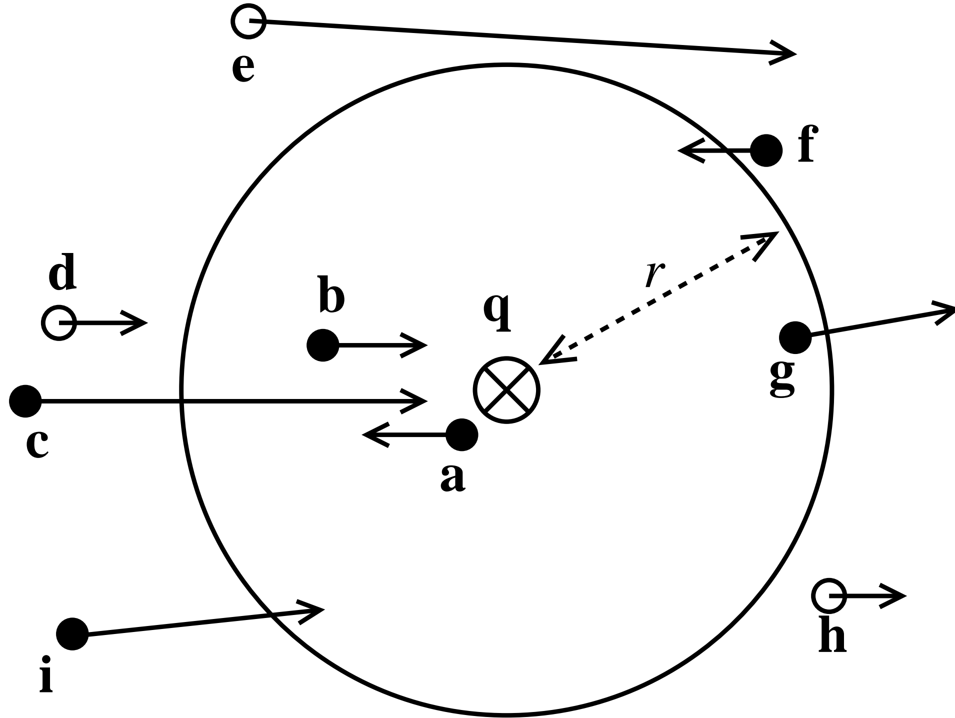


Figure 7.1: Example fuzzy set, where $\otimes$ is the query point $\mathbf{q}$, $\bullet$ indicate points in $\mathbf{q}$'s fuzzy set, and $\circ$ indicate points not in the fuzzy set.

point $\mathbf{q}$. Recall from Section 7.1, the subscript k for a semijoin $Q \ltimes_k D$ denotes the number of nearest neighbors in D to be found for every point in Q. Suppose that point $\mathbf{g}$ will leave $\mathsf{circle}(\mathbf{q}, r)$ at time $t_{\mathbf{g}}$, then the underflow event is $\mathsf{uf}(\mathbf{q}, \mathbf{g}, t_{\mathbf{g}})$. The component members of an underflow event are denoted by $\mathsf{query_pt}(\mathsf{uf}(q, p_k, t)) = q$, $\mathsf{kth_pt}(\mathsf{uf}(q, p_k, t)) = p_k$, and $\mathsf{time}(\mathsf{uf}(q, p_k, t)) = t$. For the sake of consistency with nn-event notation (described below) we define $\mathsf{other_pt}(\mathsf{uf}(q, p_k, t)) = p_k$.

An nn-event is denoted by $\mathsf{nn}(q, r, p_k, o, t)$, where q is the query point, r is the radius of $\mathsf{circle}(q, r)$, p_k is the k^{th}neighbor of q, o is the other data point that will become the new k^{th}neighbor at event time t. For example, suppose that the query for Figure 7.1 is $Q \ltimes_1 D$, that is, for each point in Q find the nearest neighbor in D. Point $\mathbf{a}$ is currently the nearest neighbor of point $\mathbf{q}$. Let the time for the next

oc-event between points $\mathbf{a}$, $\mathbf{c}$, and $\mathbf{q}$ be time $t_{\mathbf{a},\mathbf{c}}$. Also, suppose that this is the next oc-event out of all the oc-events among the points in $\mathbf{q}$'s fuzzy set. In this case, the nn-event for point $\mathbf{q}$ is $\mathsf{nn}(\mathbf{q}, r, \mathbf{a}, \mathbf{c}, t_{\mathbf{a},\mathbf{c}})$. To support fuzzy sets, the radius of the query circle is stored with the nn-event. The radius is not part of the definition of the nn-event itself, but it will be needed when the nn-event is processed. The component members of an nn-event are denoted by $\mathsf{query_pt}(\mathsf{nn}(q, r, p_k, o, t)) = q$, $\mathsf{radius}(\mathsf{nn}(q, r, p_k, o, t) = r$, $\mathsf{kth_pt}(\mathsf{nn}(q, r, p_k, o, t)) = p_k$, $\mathsf{other_pt}(\mathsf{nn}(q, r, p_k, o, t)) = o$, and $\mathsf{time}(\mathsf{nn}(q, r, p_k, o, t)) = t$.

7.2 Data Structures

The event queue E-queue is a priority queue of events (underflow and nn-events) sorted by time. It is made up of three data structures. The first is a B+-tree variant called the *nearest neighbor event B-tree* (NN-B-tree). Every point p is assumed to have an associated unique id denoted $\mathsf{id}(p)$. The NN-B-tree B+-tree is sorted on the key $\mathsf{id}(\mathsf{query_pt}(e))$ and yields the event value e (i.e., $\mathsf{id}(\mathsf{query_pt}(e)) \rightarrow e$). In addition to implementing a range tree on $\mathsf{id}(\mathsf{query_pt}(e))$, the B+-tree is augmented to implement a heap in the event times. In particular, in addition to the minimum and maximum keys, the minimum event time for a subtree in the NN-B-tree is propagated up to the root. Thus the result is a variant of a priority search tree [46]. Figure 7.2 shows an example NN-B-tree. The next event time is found by examining the root node, and returning the minimum event time in the root. To obtain the next event, the tree is traversed from its root to the leaf by following the minimum event time down the

branches of the tree. The NN-B-tree allows efficient updates based on the id of query points from Q. However, in order to efficiently perform updates using a data point id as a key, additional data structures are needed. The K-B-tree is a standard B+-tree sorted by key id(kth_pt(e)) yielding the value id(query_pt(e)) (i.e., id(kth_pt(e)) $\rightarrow$ id(query_pt(e))). The O-B-tree is a standard B+-tree sorted by key id(other_pt(e)) yielding the value id(query_pt(e)) (i.e., id(other_pt(e)) $\rightarrow$ id(query_pt(e))).

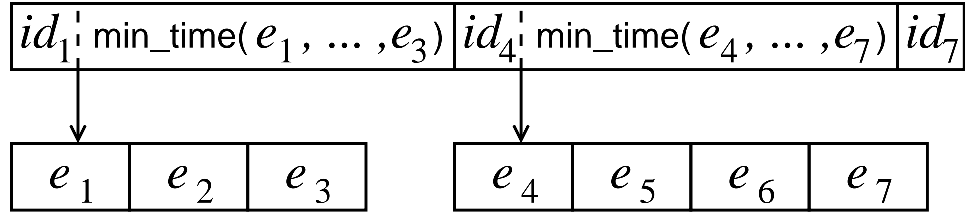


Figure 7.2: Example NN-B-tree with one root node, and two leaf nodes, where id_i = id(query_pt(e_i)).

Together, these three data structures NN-B-tree, K-B-tree, O-B-tree and their algorithms form the event queue E-queue. The algorithm to insert an event e is given in Figure 7.3. The algorithm to delete an event, given a query point q, is given in Figure 7.4. These are straight-forward since there is one and only one event for each query point in Q. The algorithm to delete all events involving a given data point p is more complicated since there may be many events involving p in the queue. The algorithm is given in Figure 7.5. First, all query points q for which p is the k^{th}neighbor are considered (line 1). The event for each q is found (line 2), then all the entries involving q are removed from the three B+-trees; O-B-tree, NN-B-tree, K-B-tree (lines 3–5). Since id(q) is unique, there is only one entry for a given q in each B+-tree. Second, all query points q are considered where p is the other point involved in q's event (line 7). Likewise, each entry for each of these query points are

deleted from the three B+-trees (lines 8–11).

procedure E-queue_Insert(e)
1. Insert id(query_pt(e)) $\rightarrow e$ into NN-B-tree.
2. Insert id(kth_pt(e)) $\rightarrow$ id(query_pt(e)) into K-B-tree.
3. Insert id(other_pt(e)) $\rightarrow$ id(query_pt(e)) into O-B-tree.

Figure 7.3: E-queue_Insert()

procedure E-queue_Delete_QueryPt(q)
1. Find entry id(q) $\rightarrow e$ in NN-B-tree.
2. Delete entry id(kth_pt(e)) $\rightarrow$ id(q) from K-B-tree.
3. Delete entry id(other_pt(e)) $\rightarrow$ id(q) from O-B-tree.
4. Delete entry id(q) $\rightarrow e$ from NN-B-tree.

Figure 7.4: E-queue_Delete_QueryPt()

procedure E-queue_Delete_All_DataPt(p)
1. **foreach** id(p) $\rightarrow$ id(q) in K-B-tree **do**
2. Find entry id(q) $\rightarrow e$ in NN-B-tree.
3. Delete id(other_pt(e)) $\rightarrow$ id(q) from O-B-tree.
4. Delete entry id(q) $\rightarrow e$ from NN-B-tree.
5. Delete entry id(p) $\rightarrow$ id(q) from K-B-tree.
6. **end foreach**
7. **foreach** id(p) $\rightarrow$ id(q) in O-B-tree **do**
8. Find entry id(q) $\rightarrow e$ in NN-B-tree.
9. Delete entry id(kth_pt(e)) $\rightarrow$ id(q) from K-B-tree.
10. Delete entry id(q) $\rightarrow e$ from NN-B-tree.
11. Delete entry id(p) $\rightarrow$ id(q) from O-B-tree.
12. **end foreach**

Figure 7.5: E-queue_Delete_All_DataPt()

To manage fuzzy sets, another set of data structures is used. The fuzzy set index (FS-index) keeps track of points in fuzzy sets, and the time the points will expire from each fuzzy set. The FS-index utilizes two B+-trees. The first B+-tree is the

FS-B-tree. It is sorted by $\mathsf{id}(q)$ yielding the value $\{q, p, t\}$ (i.e., $\mathsf{id}(q) \rightarrow \{q, p, t\}$), where $q \in Q$, $p \in D$, and t is the expiration time for p. The *expiration time* is the time when p leaves $\mathsf{circle}(q, r)$ and is no longer part of q's fuzzy set.

The other B+-tree used by the FS-index is the ID-B-tree. It is sorted by $\mathsf{id}(p)$ yielding $\mathsf{id}(q)$ (i.e., $\mathsf{id}(p) \rightarrow \mathsf{id}(q)$). It serves a similar purpose as the K-B-tree or O-B-tree for the E-queue to support deletions of data points. Together, the FS-B-tree and the ID-B-tree form the FS-index. The algorithms to insert and delete objects in the FS-index are similar to those for the E-queue, but simpler since there are only two B+-trees involved.

Two tpr indexes [58] are used by the CFS algorithm. One index is on the query circles $\mathsf{circle}(q, r)$ rather than the query points in set Q. A second tpr index used by CFS is on the set of data points D.

7.3 CFS Algorithm

The *main loop* of the event-driven algorithm processes events and updates as they occur to maintain the query result over the moving points. The main loop invokes procedures Process_Event(), Update_Fuzzy_Set(), Insert_Data_Point(), Delete_Data_Point(), Insert_Query_Point(), and Delete_Query_Point() as needed (see below). Updates are not known in advance of their occurrence. Processing continues indefinitely.

When an event on the E-queue comes due, it is dequeued and passed to procedure Process_Event() (Figure 7.6). Every query point has either an nn-event, or an underflow event associated with it in the event queue, E-queue, even if the event time is ∞.

procedure Process_Event(e)
1. Point $q \leftarrow$ query_pt(e)
2. **if** e is an nn-event **then**
3. **if** kth_pt(e) is becoming $k+1$ neighbor of q **then**
4. update the query result.
5. Get all entries for q from FS-index, and remove expired points to get fuzzy set S.
6. **if** count of expired points $>$ *expired_threshold* **then**
7. remove all expired entries for q from FS-index.
8. Enqueue_Event(S, q,radius(e),kth_pt(e))
9. **else if** e is an underflow event **then**
10. Handle_Underflow(q)

Figure 7.6: Process_Event()

This is done so that every query point and its kth neighbor can be found simply by examining the queue. For an nn-event (line 2), if other_pt(e) was not previously part of the k-neighbor-set for q (line 3), then it pushes the current k^{th} neighbor out of the set and other_pt(e) becomes the new k^{th} neighbor. This necessitates an update to the query result. The query result is updated by reporting $\langle$kth_pt$(e), q\rangle$ deleted, and $\langle$other_pt$(e), q\rangle$ inserted (line 4). For example, suppose that the query for Figure 7.1 is $Q \ltimes_1 D$, that is, for each point in Q find the nearest neighbor in D. Point **a** is currently the nearest neighbor of point **q**. The event on the queue for point **q** is nn$(\mathbf{q}, r, \mathbf{a}, \mathbf{c}, t_{\mathbf{a},\mathbf{c}})$, where time $t_{\mathbf{a},\mathbf{c}}$ is the time points **a** and point **c** will be equidistant from point **q**. When this event comes due and is processed, point **c** pushes point **a** out of the k-neighbor-set (in this case the 1-neighbor-set) of point **q**, and becomes the new nearest neighbor.

If, on the other hand, other_pt(e) was already part of the k-neighbor-set, then it simply becomes the new k^{th} neighbor, and the current k^{th} neighbor becomes the $k-1$

neighbor. For example, suppose that the query for Figure 7.1 is $Q \ltimes_2 D$, that is, for each point in Q find the 2 nearest neighbors in D. Point $\mathbf{b}$ is currently the 2nd nearest neighbor from point $\mathbf{q}$. In this example, the nn-event for point $\mathbf{q}$ is $\mathsf{nn}(\mathbf{q}, r, \mathbf{b}, \mathbf{a}, t_{\mathbf{b},\mathbf{a}})$, because point $\mathbf{a}$ will be the first to be equidistant with point $\mathbf{b}$ from point $\mathbf{q}$ before any other point in point $\mathbf{q}$'s fuzzy set. When $\mathsf{nn}(\mathbf{q}, r, \mathbf{b}, \mathbf{a}, t_{\mathbf{b},\mathbf{a}})$ comes due at time $t_{\mathbf{b},\mathbf{a}}$, point $\mathbf{a}$ becomes the new k^{th} neighbor (2nd neighbor), but point $\mathbf{b}$ stays in the 2-neighbor-set of point $\mathbf{q}$, so the query result does not change. However, since the 2nd neighbor changed, a new nn-event for point $\mathbf{q}$ must be calculated and enqueued.

The new nn-event is calculated from q's fuzzy set. The fuzzy set S for q is stored in the FS-index. All points in the FS-index that have not expired are in q's current fuzzy set (line 5). A point expires from the fuzzy set when it leaves the circle around q. If the number of expired points exceeds a certain threshold, then all expired points for q are removed from the FS-index (line 6) This keeps down the number of expired entries in the FS-index. The fuzzy set S is used to compute the next event for q (line 8) (see description of Enqueue_Event() below). An *underflow event* occurs when circle(q, r) contains less than k points (line 9). When underflow occurs, the fuzzy set must be expanded (line 10) (see description of Handle_Underflow() below).

procedure Enqueue_Event(S, q, r, p_k)

1. $S \leftarrow (S - p_k)$
2. Find the next nn-event e from among the points in S.
3. **if** kth_pt(e) will expire before e occurs **then**
4. enqueue an underflow event for q in E-queue.
5. **else** enqueue e in E-queue.

Figure 7.7: Enqueue_Event()

Enqueue_Event() (Figure 7.7), called from line 8 of Figure 7.6, computes the next event for a query point from its fuzzy set. The current k^{th} neighbor is removed from the fuzzy set S (line 1). The remaining points in fuzzy set S are each considered for the next nn-event by computing each of their next occurring order change events (oc-events) in turn (line 2). Recall from Chapter 2 that an oc-event for a data point p occurs when p moves to be at the same distance to q as its current k^{th} neighbor. The soonest oc-event becomes the next nn-event (line 5), unless circle(q, r) underflows sooner. In that case, an underflow event is enqueued instead (line 4).

procedure Handle_Underflow(q)
1. Remove all entries for q from FS-index.
2. $n \leftarrow \lceil k * circle_factor \rceil$
3. Get new set S of $n+1$ neighbors around q.
4. $r \leftarrow (\|q, s_n\| + \|q, s_{n+1}\|)/2$, where $s_n, s_{n+1} \in S$
5. Remove s_{n+1} from S.
6. Add points to S that will enter circle(q, r) during the current fuzzy-set-interval.
7. Insert points S, and their expiration times into FS-index.
8. Enqueue_Event(S, q, r, s_k), where $s_k \in S$ is the k^{th}neighbor of q.
9. Remove old circle centered at q from query point tpr tree, and insert circle(q, r).

Figure 7.8: Handle_Underflow()

Handle_Underflow() (Figure 7.8), called from line 10 of Figure 7.6, resizes the fuzzy set for a query point. The old fuzzy set needs to be removed from the FS-index, since the radius defining the expiration times for the points in the old fuzzy set will change (line 1). The circle around q is calculated to initially contain some multiple of k points. The global constant *circle_factor* > 1 is used to determine how many points to start with in a circle (line 2). An incremental distance algorithm [30]

(see Section 3.6) is used to get the $n+1$ neighbors of q using the tpr index on the data points (line 3). The $n+1$ neighbor, s_{n+1}, is used to determine the radius of the new circle. The new radius is the average of the distances from q to the nth neighbor, s_n, and q to the $(n+1)^{\text{th}}$ neighbor, s_{n+1} (line 4). Note that the Euclidean distance at the current time between two kinematic points q and p is denoted $\|q, p\|$. This technique for finding the radius helps to avoid the situation where points instantly leave the circle after it is resized. Once the radius is computed, s_{n+1} is discarded from the set S because it is outside the circle (line 5). The rest of the fuzzy set is found using an incremental within event query [65] (see Section 3.7.3) on the tpr index on the data points (line 6). At this point S contains all the points in q's new fuzzy set. The points in fuzzy set S are inserted into the FS-index along with their expiration times (line 7). The next nn-event is computed from the points in S and enqueued (line 8). Finally, the tpr index on the query circles is updated (line 9).

For example, suppose that the query for Figure 7.1 is $Q \ltimes_1 D$, that is, for each point in Q find the nearest neighbor in D. Point **a** is currently the nearest neighbor of point **q**. Also suppose that the radius of the circle is not r as in the figure, but is smaller, and suppose that an underflow event has just occurred. In other words, the radius of the circle is at the distance from point **q** that point **a** is at right now, say r_{old}. Suppose also that *circle_factor* $= 3$. When Handle_Underflow() is invoked, we get $n = k*$*circle_factor*$= 1*3 = 3$ (line 2). We then find the $n+1$, or 4 nearest neighbors to point **q** (line 3). Set S now contains points {**a, b, g, f**}. The new distance r (the large circle in Figure 7.1) is calculated to be halfway between point **g** and point **f**

from point **q** (line 4). Once r is computed, the 4th neighbor of q is removed from S leaving {**a, b, g**} (line 5). Suppose that the end of the current fuzzy-set-interval is one minute in the future. All the points that will enter circle($\mathbf{q}, r$) before the end of the current fuzzy-set-interval (e.g., within the next minute) are added to set S to give {**a,b,c,f,g,i**} (line 6). In this case, point **f** ends up back in set S, but it would not if it were moving away from the circle. The points {**a,b,c,f,g,i**}, along with their expiration times are inserted into the FS-index (line 7). They are also used to find the next nn-event (line 8). The old circle circle($\mathbf{q}, r_{old}$) is removed from the circle tpr tree and the new circle circle($\mathbf{q}, r$) is inserted (line 9).

```
    procedure Update_Fuzzy_Set()
1.  foreach circle(q, r) in the query circle tpr tree do
2.     Add new points to q's fuzzy set S that will enter
       circle(q, r) during the current fuzzy-set-interval.
3.     Find the next nn-event e from among the points in S.
4.     if e occurs before the currently enqueued event for q
5.        then replace currently enqueued event with e.
6.  end foreach
```

Figure 7.9: Update_Fuzzy_Set()

Update_Fuzzy_Set() (Figure 7.9) is invoked at the start of each new fuzzy-set-interval to update the fuzzy set for each query point. This finds all the data points that will enter query circles during the new fuzzy-set-interval segment of time. The tpr index on the query circles is scanned to get all the query circles circle(q, r) (line 1). The fuzzy set S for each q is updated by finding all the new data points entering circle(q, r) using an incremental within event query [65] (see Section 3.7.3) on the data point tpr index (line 2). The current k^{th} neighbor $s_k \in S$ is found, and then

the nn-event e from the rest of the points in S is computed (line 3). This is done by considering each point $s_i \in S, i \neq k$ for the next nn-event by computing each s_i's next oc-event with respect to s_k and q. The soonest oc-event out of all is the next nn-event e. If e occurs before the event that is currently in the event queue E-queue for q, then e replaces the one on the queue for point q (line 5).

procedure Insert_Data_Point(p)
1. **foreach** circle(q, r) with p in q's fuzzy set **do**
2. **if** p is in the k-neighbor-set of q **then**
3. Update the query result.
4. Remove q's event from the E-queue.
5. Get all entries for q from FS-index, and remove expired points to get fuzzy set S.
6. Enqueue_Event(S, q, r, s_k), where $s_k \in S$ is the k^{th}neighbor of q.
7. **else if** p introduces a sooner nn-event for q **then**
8. Replace the nn-event for q in E-queue.
9. **end if-else-if**
10. **end foreach**
11. Insert p into the data point tpr index.

Figure 7.10: Insert_Data_Point()

Insert_Data_Point() (Figure 7.10) is invoked when a new point p is added to the set of data points D in the semijoin query $Q \ltimes_k D$. The query circle tpr index is used to find all circle(q, r)'s that currently contain, or will contain p between now and the end of the current fuzzy-set-interval (line 1). In particular, this entails the performance of two operations using the query circle tpr index. The first finds all the circles that currently contain point p using a within distance $d = 0$ query [30]. The second uses an incremental within event query [65] (see Section 3.7.3) to find all the circles that will contain p before the end of the fuzzy-set-interval. Each query

circle $\mathsf{circle}(q, r)$ is processed in turn. If p is closer to a given q than the kth neighbor of q, then it is in the k-neighbor-set of q (line 2). Point q's entry in the event queue can be used to find the kth neighbor of q since both nn-events and underflow events keep track of the kth neighbor. When p is in the k-neighbor-set of q, then the current k^{th} neighbor becomes the $k+1$ neighbor. The entry involving the old k^{th} neighbor $\langle k^{\mathrm{th}}, q\rangle$ is removed from the query result and the new entry $\langle p, q\rangle$ is added (line 3). When the kth neighbor changes, the nn-event or underflow event changes as well, so the old event needs to be removed from E-queue (line 4). The new event is calculated from the fuzzy set of q (lines 5–6). If p is not in the k-neighbor-set, then it may still affect the next nn-event. If p's next oc-event occurs before the current nn-event for q, then the oc-event becomes the new nn-event, and replaces the old nn-event on the queue (lines 7–8). After all circles have been processed, the tpr index on the data points is updated (line 11).

Delete_Data_Point() (Figure 7.11) is invoked when a data point p is deleted from the set of data points D in the semijoin query $Q \ltimes_k D$. First, p is removed from the data point tpr index (line 1), and FS-index (line 2). All the circles that contain p, or would contain p during the current fuzzy-set-interval are processed in turn (line 3). These circles are found by applying an incremental distance [30] (Section 3.6), and incremental within event query [65] (Section 3.7.3) on the data point tpr index. If p is the current k^{th} neighbor, or closer to a given q than its kth neighbor, then the events and query result change (line 4). The current $k+1$ neighbor becomes the kth neighbor. After the old event for q is removed from the event queue (line

```
procedure Delete_Data_Point(p)
 1.  Remove p from the data point tpr index.
 2.  Remove all entries involving p from FS-index.
 3.  foreach circle(q, r) with p in q's fuzzy set do
 4.     if p is in the k-neighbor-set of q then
 5.        Remove q's event from the E-queue.
 6.        Get all entries for q from FS-index, and remove
           expired points to get fuzzy set S.
 7.        if number of data points in circle(q, r) < k then
 8.           Handle_Underflow(q)
 9.        else
10.           Enqueue_Event(S, q, r, s_k), where s_k ∈ S is the new k^th neighbor of q.
11.        Update query result.
12.     else if p is involved in q's enqueued event then
13.        Remove q's event from the E-queue.
14.        Get all entries for q from FS-index, and remove
           expired points to get fuzzy set S.
15.        Enqueue_Event(S, q, r, s_k), where s_k ∈ S is the new k^th neighbor of q.
16.     end if-else-if
17.  end foreach
```

Figure 7.11: Delete_Data_Point()

5), the fuzzy set S is found (line 6), and checked for underflow (line 7). Underflow results in a resizing of the fuzzy set, and a new nn-event is enqueued (line 8). If the fuzzy set does not underflow, then a new nn-event is enqueued given the new k^{th} neighbor (line 10). The result is updated by deleting the old k^{th} neighbor and inserting the new one (line 11). When p is not in the k-neighbor-set, but is involved in the nn-event for q (line 12), then the nn-event changes. In particular, the new nn-event is found from the points in q's fuzzy set S, replacing the old nn-event in the queue (lines 13–15).

Insert_Query_Point() (Figure 7.12) is invoked when a query point is inserted into Q in the semijoin query $Q \ltimes_k D$. This procedure is similar to Handle_Underflow()

procedure Insert_Query_Point(q)

1. $n \leftarrow \lceil k {*} circle_factor \rceil$
2. Get new set S of $n+1$ neighbors around q.
3. $r \leftarrow (\|q, s_n\| + \|q, s_{n+1}\|)/2$, where $s_n, s_{n+1} \in S$.
4. Remove s_{n+1} from S.
5. Add points to S that will enter circle(q, r) during the current fuzzy-set-interval.
6. Insert points S, and their expiration times into FS-index.
7. Enqueue_Event(S, q, r, s_k), where $s_k \in S$ is the k^{th}neighbor of q.
8. Report $\langle s_i, q \rangle$ inserted to result for the closest k points $s_i \in S$ to q.
9. Insert circle(q, r) into query circle tpr index.

Figure 7.12: Insert_Query_Point()

except that there are no previous entries for q in FS-index or the query circle tpr index to remove. Lines 1–7 are identical to lines 2–8 of Figure 7.8. Before finishing, the k neighbors of q are added to the query result (line 8), and the query circle is added to the index (line 9).

procedure Delete_Query_Point(q)

1. Delete the current k neighbors to q from query result.
2. Remove any entries for q from FS-index, E-queue, and the query point tpr index.

Figure 7.13: Delete_Query_Point()

Delete_Query_Point() (Figure 7.13) is invoked when a query point is deleted. It first updates the query result (line 1). This is done by applying an incremental distance query [30] (Section 3.6) on the data point tpr tree with q as the query point to determine what entries to delete. It then removes any entires involving the query point q from all the data structures (line 2).

7.4 CFS vs. CW

The CFS algorithm somewhat resembles the CW k-nn algorithm for one query point presented in [35] and the improved version in Chapter 5. However, there are significant differences. The similarity is that both approaches maintain a circular region around a query point with the constraint that it contain at least k points at all times. This filters the data points for candidates from which to select the k nearest neighbors.

The differences are in the other ways in which the circles are used. In the CW algorithm, the nn-event is computed from only those points found inside the query circle. In the CFS algorithm, points entering the circle in the near future are also considered for the next nn-event. This reduces the number of "false" nn-events that need to be changed before they occur when new candidates enter the widow of the CW algorithm. The CFS algorithm introduces the notion of fuzzy-set-intervals to limit the number of points entering the circle in the future that will be considered for the nn-event. Points entering the window in the distant future are not likely to be involved in the next nn-event. In the CW algorithm, within events are used to process points entering the window of a single query point. The CFS algorithm does not process within events as they occur. Instead, it only processes nn-events and underflow events for each query point. Within events are used in the CFS algorithm to determine when fuzzy set elements will expire.

To scale the CW algorithm to handle many query points at the same time, additional data structures would be needed to keep track of nn-events, the contents

of each query circle, the size of each query circle, and underflow. This, in addition to the sheer number of within events that would need to be queued and processed makes scaling the CW algorithm an inferior solution to the CFS algorithm.

7.5 Experiments

For the purpose of evaluating our algorithm, we scale up an existing k-nn algorithm to perform semijoin queries. We then compare the simple scaling of the previous work to the CFS algorithm.

In [35], the TP k-nn algorithm [65] was extended to support updates (presented as the ETP algorithm in [35]). Here, we scale up the ETP algorithm to do semijoins in addition to updates. We call the extension to perform semijoins the TP-semijoin (TPS) algorithm. To scale the ETP algorithm to perform semijoins, an event queue containing an nn-event for each query point is added. If for some query point q, no such event exists, then a pseudo event $\mathsf{nn}(q, p_k, p_k, \infty)$ is added to keep track of the current k^{th} neighbor p_k. When an update occurs, the event queue is scanned to determine what part of the query result, and which events need to be modified. If the set of k neighbors changes due to an update, then new neighbors and events are found using a tpr index on the data points similar to what was done in the ETP algorithm in [35]. No tpr index for the query points is needed since all the query points are in the event queue.

As discussed above (Section 7.4), the CW algorithm also presented in [35] would not scale well because there would be too many within events to process. Note

that a straight-forward scaling of the CW algorithm given in [35] can be achieved by adding an nn-event queue in addition to the within event queue. In preliminary results, scaling of the CW algorithm was found to be significantly less efficient than the TPS algorithm.

7.6 Data Sets

We use the same data sets sources as those described in Section 5.5. Table 7.1 shows the mean and standard deviation in the size of the data sets used over the entire 2 hour time interval covered by each data set. The figure also shows the average update interval (UI) for each aircraft data set.

μ	4453	9021	12690	17106
σ	330.8	680.8	962.4	1293
UI	700.7	712.8	725.1	734.6

Table 7.1: Each column corresponds to a different aircraft data set. Each row is a statistic on the data sets. Row 1 is the mean number of flights at any given time (μ). Row 2 is the standard deviation in the number of flights (σ). Row 3 is the average update interval (UI) in seconds.

Each query was performed on combinations of these subsets, not including self semijoins. Pairs of subsets were chosen at random without replacement from all possible combinations for a total of 100 joins per query. Only subsets taken from the same original data set are used in a query, so the semijoin sets are approximately the same size for each query. In other words, the number of query points is about the same as the number of data points in each semijoin query. This technique was used on both the synthetic and real data sets for comparability.

7.7 Results

Experiments were conducted in a simulation of a real-time system in which semijoin queries are maintained over time as updates occur. The experiments measured the total number of disk accesses over the duration of a query. Since we are concerned primarily with the maintenance portion of the query, the number of disk accesses used to compute the initial join result are not included. The number of disk accesses over 100 trials was averaged to yield the experiment results for a given query.

The implementation of the event queue used the generalized search tree (GiST) [28] version 0.9beta1 code. The code was compiled using gcc 2.96. The experiments were run on several VLSI 80686 CPU based machines running Linux.

The primary independent variables for comparison are the mean data set size (μ), and number of neighbors (k) to find for each query point. For the experiments where these variables do not vary, the defaults are $\mu = 9021$ for real aircraft data, $\mu = 10000$ for synthetic uniform data, and $k = 1$. Other general parameters, unless otherwise specified, are query duration of 130 seconds, disk page size of 4096 bytes, and disk cache size of 8 pages for each disk-based data structure.

Every cache page uses a least recently used (LRU) replacement policy except for the event queues. The event queues use a *Greatest Next Event* (GNE) replacement policy. GNE removes the page whose minimum next event time is the furthest in the future out of all pages in the cache. GNE worked better than LRU for small pages (e.g., 1024 bytes) and large caches (e.g., 32 pages). However, when the cache size was reduced, and the page size increased, we found nearly no difference between the

LRU and GNE policies. Therefore, LRU can be used with nearly the same results as GNE.

Parameters specific to the CFS algorithm, unless otherwise specified, are *circle_factor* = 2, *expired_threshold* = 25 events, and fuzzy-set-interval duration of 128 time units to ensure that at least one call is made to Update_Fuzzy_Set() per each 130 second query. We found these particular settings for the CFS algorithm to be nearly optimal in our experiments.

The purpose of the first experiment is to determine which algorithm, TPS or CFS, performs better in terms of disk accesses for different data sets sizes. Figure 7.14 shows the results for (a) real aircraft flight data, and (b) uniform synthetic data. Parameter k is 1. The x-axis is the average data set size (see row 1 in Table 7.1), and the y-axis is the number of disk accesses in millions (M). The points indicated by $\triangle$ symbols are the number of disk accesses for the CFS algorithm, while the $\diamond$ symbol indicates the number of disk accesses for the TPS algorithm. For the aircraft data, the CFS algorithm has 5 times fewer disk accesses than the TPS algorithm for the largest data sets tested. For the uniform synthetic data, the CFS algorithm has 10 times fewer disk accesses than the TPS algorithm for the largest data sets tested.

The purpose of the second experiment is to determine the relative performance of the CFS algorithm to the TPS algorithm when k is varied. Figure 7.15 shows the results for (a) real aircraft flight data (data set size $\mu = 9021$), and (b) uniform synthetic data (data set size $\mu = 10000$). The x-axis is k, and the y-axis is the number of disk accesses in millions (M). The points indicated by $\triangle$ symbols are the number

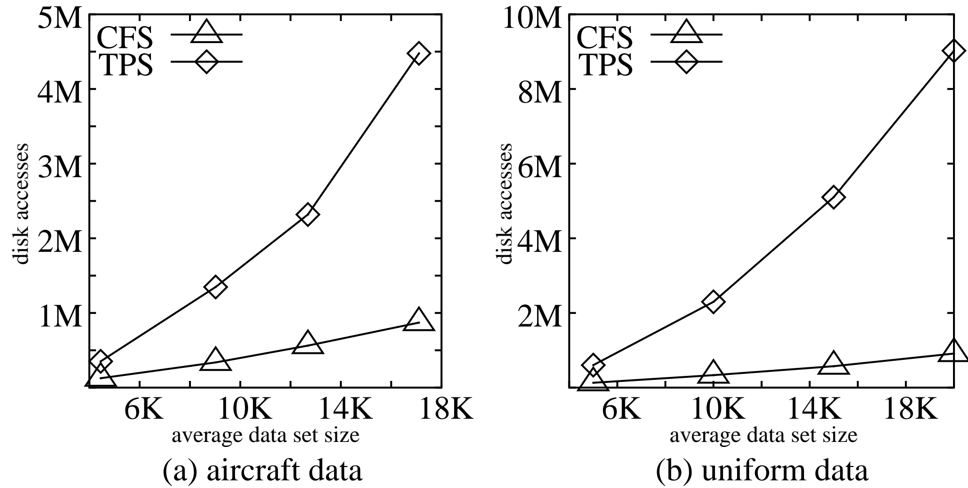


Figure 7.14: Disk accesses with respect to data set size.

of disk accesses for the CFS algorithm, while the ◇ symbol indicates the number of disk accesses for the TPS algorithm. The CFS algorithm has fewer accesses than the TPS algorithm, but the number of disk accesses for the CFS algorithm increases faster as the value of k is increased.

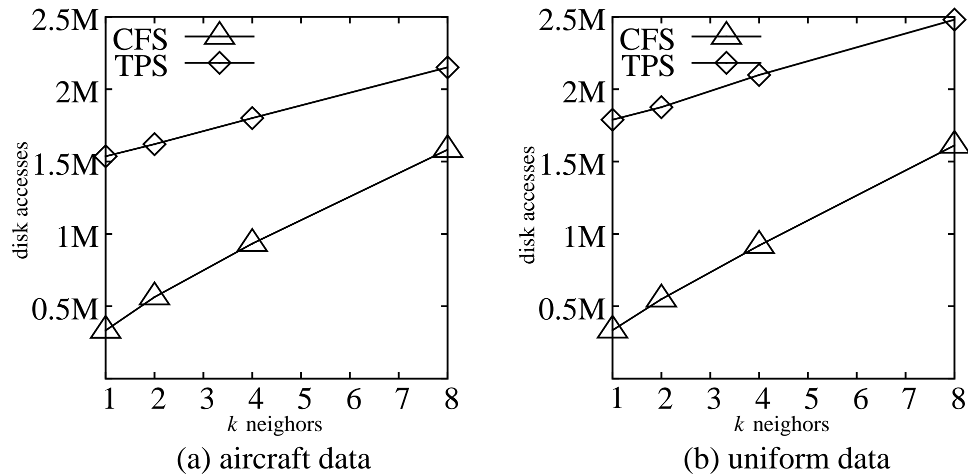


Figure 7.15: Disk accesses with respect to k with *circle_factor* = 2.

The purpose of the third experiment is to study the effect of *circle_factor* on the performance of the CFS algorithm. Figure 7.16a shows the results for real aircraft flight data (data set size $\mu = 4453$), and $k = 1$. The x-axis is the *circle_factor*, and

the y-axis is the number of disk accesses in thousands (K). The points indicated by $\triangle$ symbols are the number of disk accesses for the CFS algorithm. Although the TPS algorithm is not affected by *circle_factor*, for comparison purposes, we show the number of disk accesses ($\diamond$ symbol) for this data. From Figure 7.16a it can be seen that a *circle_factor* value of 2 yields the best performance for the CFS algorithm with $k = 1$. A *circle_factor* < 2 for $k = 1$ is not meaningful since there needs to be at least $k + 1$ points inside a query circle when it is resized. As we see, larger *circle_factor* values do lead to more disk accesses for the CFS algorithm but this is still much lower than the number of disk accesses for the TPS algorithm.

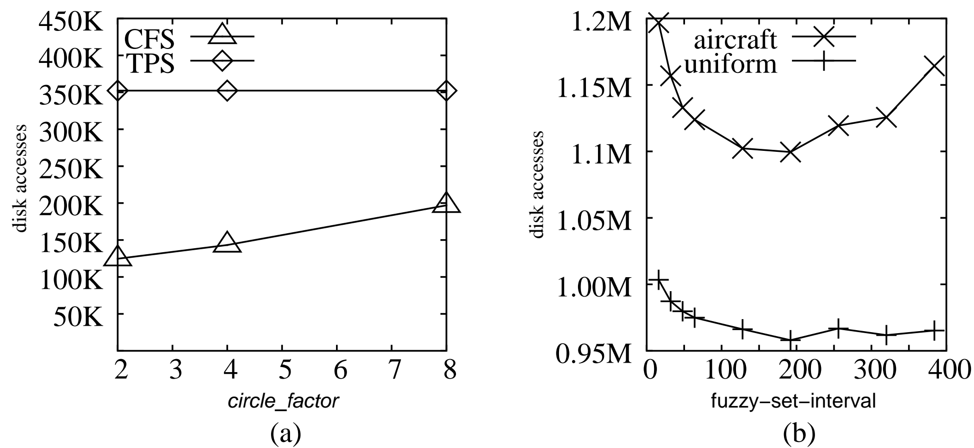


Figure 7.16: CFS algorithm parameters. (a) Disk accesses vs. *circle_factor*. (b) Disk accesses vs. fuzzy-set-interval.

Figure 7.16b shows disk accesses (y-axis) versus different values for the CFS fuzzy-set-interval parameter (x-axis) for aircraft data ($\mu = 9021$), and $k = 1$. The $\times$ symbols indicate disk accesses for aircraft data, and the $+$ symbol indicates disk accesses for uniform data. Each point is an average over 50 trials. Small values (< 128) show increased disk activity due to more frequent calls to Update_Fuzzy_Set(). Larger values (> 192) show increased disk activity due to larger fuzzy sets for each

point for the uniform data set. The reason why aircraft data does not exhibit the same performance characteristics as for uniform data for larger values of fuzzy-set-interval is unclear.

Data structure size: The implementation resulted in the following entry sizes. The tpr index entries were 32 bytes for internal node and leaf node entries. The NN-B-tree leaf node entries were 115 bytes, and internal node entries were 9 bytes. The FS-B-tree leaf node entries were 73 bytes, and internal node entries were 4 bytes. The K-B-tree, O-B-tree, and ID-B-tree leaf node entries were 8 bytes, and internal node entries were 4 bytes.

Given these numbers, we can estimate the size of the data structures under certain assumptions. Assume for query of $k = 1$ on data sets of size 20k, and page size of 4096 bytes, that 70% space utilization is achived. For each of the 20k query points, there is one event in the E-queue data structure. This gives $\lceil 20000/((4096*0.7)/115) \rceil = 803$ pages of leaf nodes in the NN-B-tree of the E-queue. and $\lceil 803/((4096*0.7)/9) \rceil = 3$ pages of internal nodes. This gives a total size of $(803 + 3) * 4069 = 3301376$ bytes on disk total for the NN-B-tree. For the K-B-tree and the O-B-tree we get $\lceil 20000/((4096*0.7)/8) \rceil = 56$ leaf node pages and 1 internal node page for a total of $(56 + 1) * 4096 = 233472$ bytes on disk for each. The total space taken by the NN-B-tree for this example is $3301376 + 262144 + 262144 = 3768320$ bytes.

To examine the FS-index, lets assume an average of 3 elements in each fuzzy set. This give $(20000*3) = 60000$ entries in the FS-index. For the FS-B-tree, this results in $\lceil 60000/((4096*0.7)/73) \rceil = 1528$ pages of leaf nodes, and $\lceil 1528/((4096*0.7)/4) \rceil = 3$

pages of internal nodes. This gives a total of $(1528 + 3) * 4096 = 6270976$ bytes on disk. For the ID-B-tree, we get $\lceil 60000/((4096 * 0.7)/8) \rceil = 168$ leaf node pages and 1 internal node page for a total of $(168 + 1) * 4096 = 692224$ bytes on disk. The total space take by FS-index for this example is $6270976 + 692224 = 6963200$ bytes.

Finally, for the tpr indexes, we get $\lceil 20000/((4096 * 0.7/32) \rceil = 224$ pages of leaf nodes, and $\lceil 224/((4096 * 0.7)/32) \rceil = 3$ pages of internal nodes. This gives a total $(224 + 3) * 4096 = 929792$ bytes on disk per tpr index.

The total disk space used for the E-queue, FS-index, and two tpr tree indices is $3768320 + 6963200 + (2 * 929792) = 12591104$ bytes in this example.

7.8 Concluding Remarks

Even with the improved performance over previous work, the number of disk accesses is still too high for the relatively small data sets to be practical. As can be seen in the example at the end of the last section, the size of the data structures is relatively small, yet the disk accesses are in the millions for small data sets over a short time interval. The main cost arises from updates to the data structures. In order to scale these algorithms up to large data sets (i.e., in the order of millions of objects) future work must focus on update efficient disk based data structures for indexing moving objects, event queues, and range trees.

In spite of these shortcomings, our experiments in Figure 7.14 show that the CFS algorithm clearly outperforms the TPS algorithm. In some cases, the difference can be as much as an order of magnitude (Figure 7.14b). The CFS algorithm is the first

algorithm of its kind to maintain spatial semijoin results on kinematic data types, over an indefinite period of time, and with no prior knowledge of the updates that will be made.

Chapter 8

Visualizing Changing Query Results for Moving Objects

8.1 Introduction

The primary focus of spatial database research has been on static spatial data [59]. Static spatial data does not change frequently, and includes keeping track of such data as buildings, roads, land use zoning, etc. The types of query operators include spatial join, nearest neighbor, and windowing. An example query is to show all the houses for sale within one kilometer of a grade school. Past research on static spatial data has primarily focused on issues of how to store and process the data to answer ad-hoc queries and display the results.

In recent years there has been an increased interest in moving object database research [49, 58, 64]. Objects in moving object databases change location frequently.

Some examples are vehicles, mobile networks, weather systems, etc. Moving object data is collected through sensors. These sensors may be located on the objects themselves (e.g. global positioning system), or the data may be collected through remote sensing (e.g. radar). Query operators include both spatial and temporal operators. An example query is to show all the school buses currently within one mile of a grade school. The dynamics of the data present different challenges in how to store the data, process queries, and display the results.

We focus on two types of moving objects. The first are *intermittently moving objects*. These objects move occasionally and then stop for relatively long periods of time. This results in a data set where relatively few objects are in motion at any one time. For example, consider cars in a parking lot. Most of the cars are parked, but a few cars may be moving at any given time. Generally, cars in the parking lot don't move for long since their drivers either find a spot to park or leave the lot. For a set of *continuously moving objects*, most of the objects are moving, and they tend to move for relatively long periods of time. We address a subset of this problem space were the motion of the moving objects are somewhat predictable at least for the near future, as opposed to random Brownian motion. For example, consider airborne aircraft. Most of the aircraft move in straight lines, or smooth curves. They tend to move at a constant speed, or constant acceleration for relatively long periods of time.

This paper presents techniques for different representation of moving object data to visualize changes in moving object queries over time using animation. Animation

provides the user with a means to visually inspect patterns or trends over time. Our goal is to generate animations that are interactive so that the user can zoom, pan, and change the playback rate during the animation. We begin by presenting technique to visualize intermittently changing data then show how these techniques may be extended to visualize continuous change.

8.2 Definitions and Notation

In a relational database, a *relation* is a table of values. Each column of a relation represents an *attribute* of a particular data type or domain (e.g. integers, strings, points, etc.). Each row in a relation is a set of related values over the attribute domains called a *tuple*. A relation's *schema* is a set of name and data type pairs, one for each attribute of a relation.

A *spatial database* is a database in which spatial attributes can be stored. A spatial database also stores non-spatial data.

Relations are updated during transactions. A *transaction* is a set of changes to a database state such that the database is left in a consistent state when the transaction completes. The database state may not change by any other means. If a transaction fails to complete, then the database reverts to its state just before the attempted transaction.

A *view* is a virtual relation defined by a database query expression. When the base relations used to define the view change, the view is reevaluated. This virtual relation may be used to define subsequent views. A *materialized view* is a view that

is stored to disk.

A relational *join* is a subset of the cross product of two relations. The cross product is filtered by a join predicate defined on the attributes of both relations. A *spatial join* involves a predicate defined on spatial attributes. Relation names are denoted by lower case letters (e.g. s). Schemas are denoted by upper case letters. To show that relation s is a collection of related tuple values over the domain of schema S we write $s(S)$. A set of attributes in a relation's schema is denoted as a series of attribute names or symbols (e.g. $\{\alpha_0, \alpha_1, ..., \alpha_{n-1}\}$). The ordering of attributes within the schema is not significant. For example, schema $S = \{\alpha, \beta\} = \{\beta, \alpha\}$ where α and β are attribute names. If $S = \{\alpha_0, \alpha_1, ..., \alpha_{n-1}\}$ where each attribute α_i $(0 \leq i < n)$ is an attribute name, then we express the domain of α_i as D_{α_i}. The domain of S is $D_S = D_{\alpha_0} \times D_{\alpha_1} \times ... \times D_{\alpha_{n-1}}$ and thus $s \subseteq D_S$.

The cross product of two relations r and s is denoted $r \times s$. The cross product of two relational domains D_R and D_S is denoted $R \times S$. The join of two relations $q(Q)$ and $r(R)$ is denoted $q \bowtie_{\mathsf{P}} r$ where P is a predicate defined on the schema attributes $Q \cup R$. At times, we write $q \bowtie r$ where P is understood. A tuple in some relation s is denoted τ_s. For some tuple $\tau_s \in s$ we denote the value of attribute α_0 in tuple τ_s as $\tau_s[\alpha_0]$. The join of two tuples $\tau_q \in q$ and $\tau_r \in r$ is their concatenation written $\tau_q\tau_r$. The domain of tuple $\tau_q\tau_r$ is $D_Q \times D_R$. Without loss of generality, we do not consider the order of attributes within a tuple to be significant, and thus $\tau_q\tau_r = \tau_r\tau_q$. If the schema's $Q \cap R \neq \emptyset$, then we assume that attributes are renamed as appropriate to avoid ambiguity when a join takes place.

There is some evidence to show that animation may be useful in detecting patterns and trends over time [39]. Alerters [10] or triggers [61] may be used to determine when a particular database state has occurred, but it may be desirable for the user to be aware of the events leading up to a particular situation. For example, a trigger can notify the user when vehicles enter a particular area of interest, but when the trigger is fired there is no general way to know what led up to that event. In some cases it may be sufficient for the user to simply monitor the display as events occur, but if the amount of time between events is very long or very short, or the user needs to do other things at the time this may not be feasible. This can make it difficult for the user to visually detect associations between events that occur over time.

One approach to get around this problem is to render spatio-temporal information in a static map [12, 39], but these can get cluttered especially if many moving objects are involved. Another approach is to capture the display output image when data is originally processed and then sequence it into an animation. Our approach is based on animation, but instead of capturing the image we capture the data from which the image is rendered to produce interactive animations. We present different methods to accomplish this depending on dynamic characteristics of both small and large sets of data.

Most, if not all, previous applications of animation to this domain have been to visualize changes in base data rather than to database query results [12, 39, 60]. In general, most previous methods render spatial data in a bitmap. One bitmap is created for each discrete time step in a series. The bitmaps are then displayed in

succession creating the animation, or in other words, animated maps. This is also known as animated cartography (See Section 3.8 for background).

One problem with bitmap animation is that if updates do not occur at a fixed rate then the perception of time relationships between events may be distorted when a fixed frame rate is used for playback. To correct this, we take the approach of time indexing the movie frames. The goal is to display each frame for a time proportional to the time between updates. For example, suppose an update occurs after one minute, then again after three more minutes. The resulting animation would have three frames. If at playback time the time between the first and second frame is one second then the time between the second and third frame is three seconds, so that it is proportional to the time between updates.

The restrictions on user interaction with the display is another drawback of bitmap animations. Our approach is to capture the minimal set of data needed to render each frame to a bitmap and render the frames when the animation plays, not when the data is generated. This allows the user to zoom in on an area of interest without loosing image quality. It also makes it easier for the user to simultaneously view multiple perspectives on the same data set.

Section 8.3 shows how to create interactive animations for small sets of moving objects. Section 8.4 then modifies those techniques to handle large sets of dynamic sample track data. Finally in Section 8.5, animation of large sets of continuously moving objects is addressed.

8.3 Small Sets of Moving Objects

To visualize changes over time, the minimal set of data needed to render the layers is saved and redisplayed in rapid succession to create an animation. Each layer in the sequence is a *layer frame*, and the whole animation sequence is a *layer sequence*. The layer frame is a data structure containing the data needed to render one animation frame. When the contents of a base relation or a query result change, the old layer frame is saved, and a new layer frame is created. When the user wants to play back the changes over time, layer frames are rendered and displayed successively in order from the oldest to the most recent.

For small sets of moving objects, we assume that any views are recomputed from scratch when the base relations or views upon which it depends are updated. Later in Sections 8.4 and 8.5, we address the animation of large sets of moving objects in which the frames are updated incrementally.

8.3.1 Fixed Update and Playback Rates

One approach is to create a new layer frame by scanning a query result (e.g., a materialized view) after each update. Each new layer frame is appended to the layer sequence. This has the advantage of decoupling the layer sequence support from the query processing.

Another approach is to build a layer sequence as the query is processed. This adds query processing overhead, but avoids scanning the query result again after it is generated. Function Process_And_Sequence() shown in Figure 8.1 shows how a layer

frame can be created during query processing. Input parameter Q is a query to be processed. Parameter $layer_seq$ is a sequence of layer frames. The query is processed incrementally. Function Process_First() processes the query until the first result tuple is generated (line 2). Subsequent result tuples are generated by Process_Next() (line 5) one at a time. Each call to Process_And_Sequence() creates a new layer frame (line 1). Minimal information needed to render the animation frame is added to the layer frame (line 4). When no more tuples are generated the new layer frame is appended to the layer sequence (line 8), and returned (line 9).

Process_And_Sequence() is called after each transaction resulting in a change to any base relation used to compute Q. Repeated calls to Process_And_Sequence() build the animation. Note that this may result in unnecessary identical frames in the case when the query result doesn't actually change as a result of the transaction. For example, if the query is to select all the red cars from a relation containing cars, then the query result will not change in the case where a single blue car is inserted into the relation. This issue goes away with the improved approach presented in Section 8.4.

Procedure Play_Sequence() (see Figure 8.2) is used to play back a layer sequence. Parameter $layer_seq$ is a sequence of layer frames. The global variable $duration$ indicates how long each frame should be displayed. The main loop iterates through each layer frame in the sequence $layer_seq$ and displays it for the given duration. Procedure Render() (line 2) clears the display, and then renders the current $layer_frame$. Procedure Render() could render one or more perspectives. These perspectives can

```
    function Process_And_Sequence(Q, layer_seq)
      begin
1.      layer_frame ← Create_New_Frame()
2.      t ← Process_First(Q)
3.      while t ≠ ∅ do
4.          Add(layer_frame, t)
5.          t ← Process_Next(Q)
6.      end while
7.      layer_seq ← layer_seq | layer_frame
8.      return layer_seq
      end
```

Figure 8.1: Function Process_And_Sequence() appends a new animation layer frame as a query is processed.

change between calls to Render() to make the animation interactive. In other words, the user may zoom or pan as the animation plays. Procedure Wait() (line 3) pauses execution of Play_Sequence() for a time period specified by *duration*. The animation playback rate can be adjusted during playback by changing the value of *duration*.

```
    procedure Play_Sequence(layer_seq)
        global var duration
      begin
1.      foreach layer_frame in layer_seq do
2.          Render(layer_frame)
3.          Wait(duration)
4.      end foreach
      end
```

Figure 8.2: Function Play_Sequence() renders an animation layer sequence.

8.3.2 Example

We will use the following example spatial join [29] query to illustrate the concepts presented in following sections. For the sake of this example, without loss of gen-

erality, the spatial data is assumed to be 2D projections and data. The techniques discussed here are not constrained by the dimensionality of the data, coordinate projections, or the screen projections used to display them.

Consider two relations $r(R)$ and $s(S)$ where schema $R = \{id, loc, type\}$, loc is a 2D point, id is a unique object identifier, and $type$ is a number. The schema of relation s is the same, $R = S$. Consider the materialized view defined in the following expression.

Example 8.3.2

$$Q = \{\tau_r \tau_s : \tau_r \in r \wedge \tau_s \in s \wedge \mathsf{Distance}(\tau_r[loc], \tau_s[loc]) \leq 2 \wedge \tau_s[type] = 1\}.$$

This example returns all pairs of objects between r and s within 2 distance units of each other, and all the object from s are of type 1.

Suppose that the initial states of r and s at time t_0 are as shown in Figures 4.1 and 4.2, respectively. Now suppose that object y is moving at a constant velocity, and object a moves and then stops as shown in Figure 4.2. Intermittent updates to the database change the current known locations of y and a.

Now, consider the spatial join expressed in Example 8.3.2. The resulting output is shown in Figure 4.3 in the first row of the table. The locations of the objects participating in the join are indicated by the ovals in Figure 4.2a. Note that although object a is within distance 2 of object x, the pair is not included in the query result because the type of x is not 1.

Now suppose at time $\mathrm{t}_0 + 1$ minutes, the s relation is updated by deleting tuple $\{y, (5, 3), 1\}$ and inserting tuple $\{y, (4, 3), 1\}$, and the r relation is updated by deleting tuple $\{a, (2, 2), 1\}$ and inserting tuple $\{a, (1, 1), 1\}$. The update results in the recom-

puted spatial join shown in the second row of Figure 4.3, and graphically by the oval in Figure 4.2b. Now suppose after 3 more minutes at time $t_0 + 4$ an update changes object y's location from $(4, 3)$ to location $(1, 3)$. The join result after the update are shown in the third row of Figure 4.3, and corresponding ovals in Figure 4.2c.

8.3.3 Variable Update and Playback Rates

The algorithms presented so far work well if updates to the base relations occur at fixed intervals in time. If the transactions that update the base relations occur at random intervals, then the perception of the temporal relationships between events may be distorted during playback using these algorithms. This is because the amount of time between the creation of each frame may be different from frame to frame. Procedure Play_Sequence() does not take the amount of time between updates into account. To support random time intervals between updates, function Process_Variable_Rate() replaces function Process_And_Sequence(), and procedure Play_Variable_Rate() replaces procedure Play_Sequence(). These procedures are designed to display frames for a period of time proportional to the time between updates.

Process_Variable_Rate() (see Figure 8.3) is similar to Process_And_Sequence(). An additional input parameter, τ_times, is a sequence of transaction times. Each element in τ_times has a one-to-one correspondence with each $layer_frame$ in the $layer_seq$. In other words, the first element in τ_times corresponds to the first element in $layer_seq$, and so forth. The time of the last transaction is appended to

τ_times (line 8), and returned along with the layer sequence (line 9).

```
      function Process_Variable_Rate(Q, time, layer_seq, τ_times)
        begin
1.        layer_frame ← Create_New_Frame()
2.        t ← Process_First(Q)
3.        while t ≠ ∅ do
4.          Add(layer_frame, t)
5.          t ← Process_Next(Q)
6.        end while
7.        layer_seq ← layer_seq | layer_frame
8.        τ_times ← τ_times | time
9.        return layer_seq and τ_times
        end
```

Figure 8.3: Function Process_Variable_Rate() saves a transaction time for each layer frame.

Procedure Play_Variable_Rate() (see Figure 8.4) uses the τ_times data gathered by Process_Variable_Rate() to determine the duration of animation frames during playback. As an example, consider the query given in Section 8.3.2. An update transaction occurs after one minute and the next one occurs after another three minutes. The resulting animation has three frames. If the animation is played back so that the duration of the first frame is half a second, then it would follow that the duration of the second frame is 1.5 seconds. At this rate, playback is 120 times faster than real time. The variable *factor* controls playback rate. A value greater than 0 but less than 1 is faster than real time. A value greater than 1 is slower than real time. If $factor = 1$, then the playback will be close to real time plus some added time for processing overhead. A factor of 2 means it takes twice as long to play the animation, and a factor of 0.5 is half as long. The algorithm could be made more precise by subtracting the processing overhead time from the computed *duration*

value. For simplicity, processing overhead is not considered here.

The main loop of the procedure iterates through the transaction times using functions first() and rest() (lines 1 and 2). The function first() returns the first element in a sequence. Function rest() returns all but the first element. The *duration* of each frame is computed as the time difference between the current transaction time and the next multiplied by $factor$ (line 3). The layer frame is then rendered (line 4), and the procedure pauses for a moment (line 5) before moving on. When there are no more transaction times, the last layer frame is displayed (line 9) and the procedure exits. Variable $factor$ is shown as a a global variable to show it may be changed on the fly.

```
    procedure Play_Variable_Rate(layer_seq, τ_times)
        global var factor
      begin
1.      cur_time ← first(τ_times)
2.      foreach next_time in rest(τ_times) do
3.        duration ← (next_time − cur_time) ∗ factor
4.        Render(first(layer_seq))
5.        Wait(duration)
6.        layer_seq ← rest(layer_seq)
7.        cur_time ← next_time
8.      end foreach
9.      Render(first(layer_seq))
      end
```

Figure 8.4: Procedure Play_Variable_Rate()

8.3.4 Variable Update Rate and Fixed Playback Rate

At times, it may be desirable to export a bitmap animation for use in a web page, or for import into another application. Procedure Play_Variable_Rate() supports variable time intervals between transactions; however, variable frame rate playback is not very conventional. Many standard animation formats use a fixed frame rate.

Function Convert() (see Figure 8.5) converts a variable rate layer frame sequence to a bitmap animation with a fixed frame rate. Input parameter *duration* is the desired duration of each frame in the output bitmap animation. Parameter *factor* controls how much faster or slower than real time the output animation will appear to be. Basically, the algorithm sees how many times it can chop up each variable length input frame into fixed length output frames. Since the given output *duration* may not evenly divide into the frame duration of a layer frame, the interval must be rounded off. The difference is saved and added into the next layer frame duration time.

In Convert(), the variable *animation* (line 1) is the output bitmap animation. Variable *remainder*(line 2) is the amount of time left over from the display of the last layer frame. The *cur_time* (line 3) is the update transaction time for the current layer frame being processed. The sequence of transaction time is artificially extended by one more value (line 4) so the last frame will have a duration. In this case the duration of the last input frame equal to the duration of the first input frame. The main loop iterates on the transaction time sequence (line 5). Each iteration of the loop processes one variable length input frame producing zero or more fixed length

output frames. It is possible that a frame may be dropped if the duration of an input frame is less than the duration of an output frame. For simplicity, this case is assumed rare and is not given any consideration here. In each iteration of the loop, the display time of a layer frame is calculated using the given $factor$ value (line 6). This is then chopped up into a number of equal duration output animation frames (line 7), and any leftover time is saved for the next iteration (line 8). If the leftover time is greater than half the duration of a bitmap animation frame the number of output frames is rounded up (lines 9 - 12). The bitmap animation frames are then rendered and appended to the output animation (lines 13 - 15). The algorithm then moves on to the next layer frame (lines 16 - 17). The new fixed frame duration bitmap animation is returned (line 19).

Figure 8.1 shows a trace of function Convert() on some example input. To see how this works, consider the example from Section 8.3.2. In the example the first update transaction occurs after one minute and the second update occurs after another three minutes. If time is measured in milliseconds, then the input parameter τ_times is the sequence (0, 60000, 240000). Let input parameter $duration = 62.5ms$ (16 frames per second), and parameter $factor = 0.01$. Parameter $factor = 0.01$ will produce an animation 100 times faster than real time. The left column of Figure 8.1 shows at what line number the action for that row was performed. A number in a cell indicates a variable set to a new value. Boolean values indicate an expression evaluation result. Variables not affecting the control flow are not shown.

```
function Convert(layer_seq, τ_times, duration, factor)
  begin
1.    animation ← NULL
2.    remainder ← 0
3.    cur_time ← first(τ_times)
4.    τ_times ← τ_times | (last(τ_times)
                  + first(rest(τ_times)) − first(τ_times))
5.    foreach next_time in rest(τ_times) do
6.       delta ← ((next_time − cur_time) ∗ factor)
                + remainder
7.       frame_count ← floor(delta ÷ duration)
8.       remainder ← mod(delta, frame) ∗ duration
9.       if remainder > duration ÷ 2 then
10.         frame_count ← frame_count + 1
11.         remainder ← remainder − duration
12.      end if
13.      for i ← 1 to frame_count do
14.         animation ← animation | Render(first(layer_seq))
15.      end for
16.      layer_seq ← rest(layer_seq)
17.      cur_time ← next_time
18.   end foreach
19.   return animation
  end
```

Figure 8.5: Procedure Convert() converts a variable rate layer frame sequence to a fixed rate bitmap animation.

8.4 Large Sets of Intermittently Moving Objects

This section addresses the problem of animation of larger data sets where relatively few tuples are updated in any given transaction. This is a reasonable assumption in the case where objects move intermittently, which means that only a few objects move at any given time or that updates of the object's positions are infrequent regardless of how much they move. The animation creation algorithms described in Section 8.3 may be prohibitively slow for large data sets since they rely on the

`line num`	*next _time*	*cur _time*	*delta*	*remainder*	*frame _count*	*remainder > duration ÷2*
1				0		
3		0				
5	60000					
6			600			
7					9	
8				37.5		
9						true
10					10	
11				-25		
17		60000				
5	240000					
6			1775			
7					28	
8				25		
9						false
17		240000				
5	300000					
6			625			
7					10	
8				0		
9						false
17		300000				

Table 8.1: Example trace of procedure Convert

complete reevaluation of the query even when just one tuple is updated. Location information in this section is assumed to be sample track data.

The *heuristic of inertia* [25] assumes that each update to a large data set is small. It is a basic assumption for many incremental view maintenance algorithms [9, 25]. Incremental view maintenance is a class of techniques developed to update view

query results efficiently based on changes made to base relations during a transaction. These base relation changes are used to compute the net change to the materialized view, and are stored in an axillary tables, i_v for inserts, and d_v for deletions for some materialized view v (see Section 3.5 for more information on incremental view maintenance).

A differential table is an auxiliary relation associated with a relation or intermediate query result. A differential table contains all the tuples inserted into, or all the tuples deleted from, a relation during a transaction. Before committing the transaction and updating the materialized view on disk, the differential tables can be used to generate the next layer frame in an animation sequence. The differential tables are small, and can be used to generate the new frames without having to endure costly query computation in the process.

The function Process_Incremental() shown in Figure 8.6 builds the next layer frame in a sequence from differential tables. This function is invoked after each transaction that results in a change to the base relations or views upon which some view v is defined. If there is no net change as a result of the transaction then nothing is done (line 1) and the sequences are returned unchanged. This avoids the creation of consecutive duplicate layer frames in the sequence.

The last layer frame in the sequence is then copied (line 2). This can be a shallow copy duplicating the containing data structure (e.g., a binary tree), with references to the same spatial data (e.g., point coordinates) to save memory. The minimal data needed for rendering is then added for each tuple in the insert differential table

(lines 3 - 5). The data from previously inserted tuples is removed for tuples in the delete differential table (lines 6 - 8). To support efficient $O(log(n))$ deletion it may be desirable to store the layer frame data in a tree data sorted by a key attribute in tuple t, or organized spatially (e.g. k-d-tree, quad-tree, etc.). The new layer frame and transaction time are appended to their respective sequences (lines 9 - 10), and returned (line 12).

The resulting layer frame sequence can be played using Play_Variable_Rate() shown in Figure 8.4.

```
     function Process_Incremental(i_v, d_v, time, layer_seq, τ_times)
       begin
 1.      if (i_v ≠ ∅ ∨ d_v ≠ ∅) ∧ i_v ≠ d_v then
 2.        layer_frame ← Copy(last(layer_seq))
 3.        foreach tuple t in i_v do
 4.          Add(layer_frame, t)
 5.        end foreach
 6.        foreach tuple t in d_v do
 7.          Remove(layer_frame, t)
 8.        end foreach
 9.        layer_seq ← layer_seq | layer_frame
10.        τ_times ← τ_times | time
11.      end if
12.      return layer_seq and τ_times
       end
```

Figure 8.6: Function Process_Incremental() copies the previous layer frame and applies the changes from differential tables i_v, and d_v.

8.5 Large Sets of Continuously Moving Objects

For large sets of continuously moving objects we assume a kinematic representation. Using this representation, all objects may be moving, but the coefficients of the

functions describing their motion need only be updated occasionally. For example, aircraft may move between way points requiring updates to their function descriptions only when they change course. Assuming the heuristic of inertia, the same function Process_Incremental() (Figure 8.6) can be used to build the kinematic layer frame sequence.

To play back the sequence, procedure Play_Kinematic() (Figure 8.7) is used. In the procedure, the input parameter *duration* is the time between output animation frames. Parameters *layer_seq*, and *τ_times* are as before. Global parameter *factor* plays the same role as in Play_Variable_Rate() controlling the rate of change in the data relative to real time (line 6). The procedure iterates through each layer frame in *layer_seq* and renders each one zero or more times depending on the values of *duration*, *factor*, and the actual time between transactions. Procedure Kinematic_Render() (line 4) evaluates each kinematic function stored as part of the given layer frame with respect to the given *time* to find where the object is at that moment and renders it. The *time* is incremented after every call to Kinematic_Render() so the output animation frame is different for each call even when the layer frame may be the same for two consecutive calls. This results in a smooth animation regardless of how fast or slow the animation is playing.

8.6 Conclusion

In this chapter we presented algorithms to visualize changing query results over continuously moving objects using animation. We implemented the algorithms for

```
procedure Play_Kinematic(layer_seq, τ_times,
                                         duration)
      global var factor
    begin
1.    time ← first(τ_times)
2.    foreach next_time in rest(τ_times) do
3.      while time < next_time do
4.        Kinematic_Render(first(layer_seq), time)
5.        Wait(duration)
6.        time ← time + (duration * (1/factor))
7.      end while
8.      layer_seq ← rest(layer_seq)
9.    end foreach
    end
```

Figure 8.7: Function Play_Kinematic()

large sets of moving objects. The implementation was in JAVA running as an applet in a web browser accessing a remote server. The algorithms functioned as expected. We could easily zoom, pan and vary the speed of the animation during playback. We also noted that slowed animations of continuously moving objects represented by kinematic data types did appear much smoother than for the sample representations.

No empirical studies were performed. It is our belief that kinematic data types combined with incremental view maintenance can greatly reduce network load. The continuously moving object visualization algorithms enable this representation to be visualized as an animation on a client. In the future we would like to combine these components into a coherent system and compare the results experimentally with more conventional approaches.

Chapter 9

Conclusion

Our hypothesis is that algorithms for the maintenance of spatial queries on kinematic point data types can be developed to support updates to base relations as time advances that are more efficient than straight forward adaptations of previous work. To support this hypothesis we presented algorithms to maintain k-nearest neighbor (Chapter 5), spatial join (Chapter 6), and semijoin queries (Chapter 7) on kinematic points. We compared these algorithms experimentally using both synthetic and real aircraft data.

Experiments in Chapter 5 show that the new continuous windowing (CW) k-nearest neighbor algorithm clearly outperformed the ETP algorithm. The ETP algorithm is the Tao and Papadias TP algorithm [65] extended by us to support updates. The strategy of the CW algorithm is based on the observation that w-events are fundamentally cheaper to process than nn-events. The CW algorithms filters the points considered for the nn-event using w-events to maintain the set of

points close to the query point. The ETP algorithm uses a TPR-tree index on the data points to find each subsequent nn-event. Although the CW approach processes more events overall, the cost in the CW algorithm to maintain the event queue when the base relations are updated is cheaper than updating the TPR-tree index used by the ETP algorithm. Additional support for underflow and dynamic query window sizing was added to the CW algorithm that was not included in the original CW algorithm presented in [35].

The spatial join algorithm presented in Chapter 6 is a generalization of the event-based query algorithms to support updates that are presented in Chapter 5. Two new approaches were compared that differ in the number of events placed on the queue. The All-Event (AE) approach can be thought of as an extension of the continuous spatial join (CSJ) algorithm presented in [65] to support updates in the same way that the ETP algorithm in Chapter 5 is an extension of the TP algorihtm to support updates in k-nearest neighbor queries. The AE approach stores all within events to occur in the near future in a priority event queue. The more novel Next-Event (NE) approach only stores the next event to occur for each query point in the event queue. The time period considered for future events is limited to the current event generation cycle. When the event generation cycle is short, the AE approach results in fewer disk accesses than the NE algorithm. When the event generation cycle is long, the NE approach results in fewer disk accesses because the size of the event queue is smaller. Both the AE and NE approaches outperform a simpler adaptation of the CSJ algorithm to support updates by up to two orders of magnitude.

In Chapter 7 the continuous fuzzy set (CFS) algorithm is presented as a means to maintain spatial semijoin queries. This was compared experimentally to a scaled up version of the ETP algorithm from Chapter 5, called the time-parameterized semijoin (TPS) algorithm. Although the CFS algorithm clearly outperforms the TPS algorithm, experimental results are not promising for the scalability of the implementation of this algorithm to larger data sets.

Updates to the TPR-tree and B+-tree based indexes are costly. Future work should focus on update efficient supporting data structures. For example, the design of the event queue was changed from the implementation of the algorithms in Chapter 6 to the implementation in Chapter 7 because the implementation in Chapter 7 was found to be more efficient. Now that algorithms to support the maintenance of spatial queries on point kinematic data types have been developed, more update efficient disk based data structures should help improve the performance of these algorithms and improve their scalability.

The Internet Spatial Spreadsheet (ISS) presented in Chapter 4, and the visualization techniques presented in Chapter 8, round out our work by presenting a context and motivation for our moving object query algorithms.

Appendix A

View Maintenance Proof

A.1 Notation

A schema is a set of attribute names denoted by an uppercase letter. The ordering of attributes within the schema is not significant. For example, schema $S = \{\alpha, \beta\} = \{\beta, \alpha\}$ where α and β are attribute names. Relation names are denoted by lower case letters. To show that relation s is a collection of related tuple values over the domain of schema S we write $s(S)$. For a given relation, the name of its schema is the same as the name of the relation, except that it is an uppercase letter, unless otherwise stated. If $S = \{\alpha_0, \alpha_1, ..., \alpha_{n-1}\}$ where each attribute α_i $(0 \leq i < n)$ is an attribute name, then we express the domain of α_i as D_{α_i}. The domain of S is $D_S = D_{\alpha_0} \times D_{\alpha_1} \times ... \times D_{\alpha_{n-1}}$ and thus $s \subseteq D_S$. For some tuple $\tau_s \in s$ we denote the value of attribute α_0 in tuple τ_s as $\tau_s[\alpha_0]$.

The join of two relations $q(Q)$ and $r(R)$ is denoted $q \bowtie_{\mathsf{P}} r$ where P is a predicate defined on the schema attributes $Q \cup R$. Sometimes we write $q \bowtie r$ where P is

understood. The concatenation of two tuples $\tau_q \in q$ and $\tau_r \in r$ is written $\tau_q\tau_r$. The domain of tuple $\tau_q\tau_r$ is $D_Q \times D_R$. Without loss of generality, we do not consider the order of attributes within a tuple to be significant, and thus $\tau_q\tau_r = \tau_r\tau_q$. If the schemas $Q \cap R \neq \emptyset$, then we assume that attributes are renamed as appropriate to avoid ambiguity when a tuple $\tau_q\tau_r$ is joined.

Let $r(R)$ and $s(R)$ be relations. Let the function $\mathsf{insert}(r, s)$ return all the tuples from relation r and relation s as one relation (duplicates allowed). Let the function $\mathsf{delete}(r, s)$ return the tuples found in relation r that have no equivalent counterpart among the tuples of relation s. In the case of duplicates, only one duplicate tuple is deleted for r for each match found in s. As a means of shorthand, let $\mathsf{insert}(r, s) \equiv r \uplus s$ and $\mathsf{delete}(r, s) \equiv r - s$. The insertion of a tuple τ_r into relation s is written $s \uplus \tau_r$. The size or number of tuples contained in a relation r is denoted as $|r|$. For example, suppose relation $r = \{(\text{a,b}),(\text{a,b})\}$ and relation $s = \{(\text{a,b})\}$ then $r \uplus s = \{(\text{a,b}),(\text{a,b}),(\text{a,b})\}$, $r - s = \{(\text{a,b})\}$, and $|r| = 2$.

A differential table is an auxiliary relation associated with a base relation or intermediate query result. A differential table contains all the tuples inserted into, or all the tuples deleted from, a relation during a transaction. If s is the state of a relation before some transaction Φ, then s' denotes the state of a relation after transaction Φ. Symbol i_s denotes the relation of tuples inserted into s during transaction Φ, and d_s is the relation of tuples deleted during the transaction. Relations i_s and d_s are the differential tables for relation s.

A.2 Incremental Update of Spatial Join Views

The view maintenance algorithm is described in terms of insert, delete and join operations on relations.

Here are some assumptions. A tuple can be deleted and then reinserted during the same transaction. A join operation is a binary operation in that it joins two input relations. Joins of more than two relations can be achieved by nesting operations, e.g. $((r \bowtie s) \bowtie t)$. It is assumed that no attempt will be made to delete a tuple from a base relation that is not already in the relation, in other words $(d_s \subseteq s)$. Relations may contain duplicate tuples. If a relation has no specified key or discriminator, then the value of the whole tuple is assumed to be the tuple discriminator. It is assumed that the old version of each base relation (e.g. state of joined tables the last transaction) is available. It is also assumed that differential tables i_s and d_s are available for each input relation.

The differential tables used here are similar to *hypothetical relations* [67]. For some relation r, the intersections $i_r \cap r$ and $i_r \cap d_r$ are not necessarily empty. This is in contrast to the disjoint property of the tables described in [9]. The difference between hypothetical relations and our differential tables is that no extra attributes are required. The schema of each differential table is the same as its associated base relation. Another characteristic of this algorithm is that it does not require the use of keys, or any addition bookkeeping information to accompany base relations or query results.

Base relation update: Let relation i_s be the tuples inserted into relation s

during transaction Φ. The insertion update to s is expressed as $s' = s \uplus i_s$. Let relation d_s be all tuples deleted from relation s during the same transaction. The deletion update to s is expressed as $s' = s - d_s$. By combining these two expressions we get $s' = (s \uplus i_s) - d_s$. The parentheses show proper precedence needed in the case that a tuple is inserted and deleted during the same transaction.

View update: Now, consider all views v in a database system. An update v resulting in v' can be expressed in terms of differential tables $v' = (v \uplus i_v) - d_v$. After the base relations in a system are updated by a transaction, the differential tables are then computed for each view. Views depending only on base relations are computed first followed by views depending on other views where their dependencies have already been computed. The dependencies must not be circular. After all differential tables are computed for all views, the views are updated by inserting tuples contained in the i_v differential tables followed by deleting the tuples contained in the d_v differential tables. Once this is finished, the differential tables are cleared and the system is ready for another transaction. This process allows for incremental updates of nested views.

Join update: Let $j' = l' \bowtie r$ be a join operation update after relation l is updated during some transaction Φ. By substitution this expression can be rewritten in terms of the relation l's differential tables and the state of l before the transaction as $j' = ((l \uplus i_l) - d_l) \bowtie r$. The join operation is distributive over the $\uplus$ and $-$ operations, so this expression can be rewritten as $j' = ((l \bowtie r) \uplus (i_l \bowtie r)) - (d_l \bowtie r)$. Substituting j for $(l \bowtie r)$ the expression becomes $j' = (j \uplus (i_l \bowtie r)) - (d_l \bowtie r)$.

The $(i_l \bowtie r)$ term is the expression for the insert differential table of relation j, and the term $(d_l \bowtie r)$ is the expression for the delete differential table for relation j. This gives us a simple expression for the case when only one of the joined relations changes during a transaction.

The case when both of the joined relations change during a transaction is more complicated. Here we derive an expression to compute the differential tables involving changes in both of the joined relations. To create an incremental view maintenance algorithm for join, the following expressions are used.

Exp. A.2.1 $\quad l' = ((l \uplus i_l) - d_l)$

Exp. A.2.2 $\quad r' = ((r \uplus i_r) - d_r)$

Exp. A.2.3 $\quad j' = l' \bowtie r'$

Exp. A.2.4 $\quad j' = ((l \uplus i_l) - d_l) \bowtie ((r \uplus i_r) - d_r)$

Expressions A.2.1 and A.2.2 represent the update of two relations after a transaction. Expression A.2.3 represents a join of those relations in terms of the state of the joined relations after a transaction. By substitution of expression A.2.1 and A.2.2 into expression A.2.3, the join operation can be expressed in terms of the states of the joined relations before the transaction, and the differential tables. This is shown in expression A.2.4. To make expression A.2.4 useful, the joins are distributed over the relation addition ($\uplus$) and subtraction ($-$) operations to produce expression A.2.5. A proof of the correctness of expression A.2.5 is given in Section A.2.1.

Exp. A.2.5

$$j' = (j \uplus (((((i_l \bowtie r) \uplus (l \bowtie i_r)) \uplus (i_l \bowtie i_r)) - (i_l \bowtie d_r)) - (i_r \bowtie d_l)))$$
$$- (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

The subexpressions of expression A.2.5 form the basis for an algorithm to calculate the join view differential tables i_j and d_j (see expressions A.2.6 and A.2.7 below).

Exp. A.2.6

$$i_j = ((((i_l \bowtie r) \uplus (l \bowtie i_r)) \uplus (i_l \bowtie i_r)) - (i_l \bowtie d_r)) - (i_r \bowtie d_l)$$

Exp. A.2.7

$$d_j = (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

A.2.1 Correctness

In this section we prove the correctness of expression A.2.5. Formal definitions of the addition, subtraction and join of relations are presented. Following the definitions, we present six rewrite rules used in the proof of expression A.2.5.

Definition A.2.1.1 *Addition of relations*: Given relations $s(S)$ and $t(S)$, $s \uplus t = \{\tau : \tau \in s \vee \tau \in t\}$, duplicates allowed.

Definition A.2.1.2 *Subtraction of relations*: Given relations $s(S)$ and $t(S)$, $s - t = \{\tau : \tau \in s \wedge \tau \notin t\}$ where only one tuple element of s is removed for each matching element in t.

Definition A.2.1.3 *Join of relations*: Given relations $s(S)$ and $t(T)$ and predicate P defined on a subset of $S \cup T$, $s \bowtie_{\mathsf{P}} t = \{\tau_s \tau_t : \tau_s \in s \wedge \tau_t \in t \wedge \mathsf{P}(\tau_s, \tau_t)\}$.

In the following, rewrite rules s, t and v are relations and $\bowtie$, $-$, and $\uplus$ are binary operations on relations. A sample proof of a rewrite rule is found in Appendix A.3.

Rule 1: *Distribution of join over subtraction*

$$(s - t) \bowtie v \longleftrightarrow (s \bowtie v) - (t \bowtie v)$$

Rule 2: *Distribution of join over addition*

$$(s \uplus t) \bowtie v \longleftrightarrow (s \bowtie v) \uplus (t \bowtie v)$$

Rule 3: *Commutativity of join*

$$s \bowtie t \longleftrightarrow t \bowtie s$$

Rule 4: *Associativity of subtraction*

$$s - (t \uplus v) \longleftrightarrow (s - t) - v$$

Rule 5: *Associativity of addition*

$$(s \uplus t) \uplus v \longleftrightarrow s \uplus (t \uplus v)$$

Rule 6: *Associativity of subtraction of subset relations*

$$(s \uplus t) - v \longleftrightarrow s \uplus (t - v) \text{ iff } v \subseteq t \vee s \cap v = \emptyset$$

Rule 7: *Commutativity of addition*

$$s \uplus t \longleftrightarrow t \uplus s$$

By successively applying rules 1, 2 and 3 to expression A.2.4, the join operators are pushed down to the lowest possible level. Rules are then applied to rearrange the join terms. The rewrite rules used for each step are indicated by the numbers inside curly braces at the end of each line. For example, {3,5} indicates one or more applications of rule 3 followed by one or more applications of rule 5.

Proof:

$$j' = ((l \uplus i_l) - d_l) \bowtie ((r \uplus i_r) - d_r)$$

{expression A.2.4}

$$= ((l \uplus i_l) \bowtie ((r \uplus i_r) - d_r)) - (d_l \bowtie ((r \uplus i_r) - d_r))$$

{1}

$$= (((r \uplus i_r) \bowtie (l \uplus i_l)) - (d_r \bowtie (l \uplus i_l)))$$

$$- (((r \uplus i_r) \bowtie d_l) - (d_r \bowtie d_l))$$

{3,2}

$$= (((r \bowtie (l \uplus i_l)) \uplus (i_r \bowtie (l \uplus i_l))) - ((l \bowtie d_r) \uplus (i_l \bowtie d_r)))$$

$$- (((r \bowtie d_l) \uplus (i_r \bowtie d_l)) - (d_r \bowtie d_l))$$

{3,2}

$$= ((((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- ((l \bowtie d_r) \uplus (i_l \bowtie d_r))) - (((r \bowtie d_l) \uplus (i_r \bowtie d_l)) - (d_r \bowtie d_l))$$

{3,2}

$$= (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- (((l \bowtie d_r) \uplus (i_l \bowtie d_r)) \uplus (((r \bowtie d_l) \uplus (i_r \bowtie d_l)) - (d_r \bowtie d_l)))$$

{4}

- $d_r \subseteq r$ is true by assumption.
- $d_r \subseteq r \Rightarrow (d_r \bowtie d_l) \subseteq (r \bowtie d_l)$

$$\Rightarrow (d_r \bowtie d_l) \subseteq ((r \bowtie d_l) \uplus (i_r \bowtie d_l))$$

so

$$j' = (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- ((((l \bowtie d_r) \uplus (i_l \bowtie d_r)) \uplus ((r \bowtie d_l) \uplus (i_r \bowtie d_l))) - (d_r \bowtie d_l))$$

{6}

$$= (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- (((i_l \bowtie d_r) \uplus ((l \bowtie d_r) \uplus ((r \bowtie d_l) \uplus (i_r \bowtie d_l)))) - (d_r \bowtie d_l))$$

{7,5}

$$= (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- (((i_l \bowtie d_r) \uplus ((i_r \bowtie d_l) \uplus ((r \bowtie d_l) \uplus (l \bowtie d_r)))) - (d_r \bowtie d_l))$$

{7,5}

$$= (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- ((((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)) \uplus ((r \bowtie d_l) \uplus (l \bowtie d_r))) - (d_r \bowtie d_l))$$

{5}

- $d_r \subseteq r$ is true by assumption.
- $d_r \subseteq r \Rightarrow (d_r \bowtie d_l) \subseteq (r \bowtie d_l)$

  $\Rightarrow (d_r \bowtie d_l) \subseteq ((r \bowtie d_l) \uplus (l \bowtie d_r))$

so

$$j' = (((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- (((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)) \quad \uplus (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l)))$$

{6}

$$= ((((l \bowtie r) \uplus (i_l \bowtie r)) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

$$\{4\}$$

$$= (((l \bowtie r) \uplus ((i_l \bowtie r) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r))))$$

$$- ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

$$\{5\}$$

- $d_l \subseteq l \wedge d_r \subseteq r$ is true by assumption.
- $d_l \subseteq l \Rightarrow (i_r \bowtie d_l) \subseteq (i_r \bowtie l)$
- $d_r \subseteq r \Rightarrow (i_l \bowtie d_r) \subseteq (i_l \bowtie r)$
- $(i_r \bowtie d_l) \subseteq (i_r \bowtie l) \wedge (i_l \bowtie d_r) \subseteq (i_l \bowtie r)$

  $\Rightarrow ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)) \subseteq ((i_l \bowtie r) \uplus (l \bowtie i_r))$

  $\Rightarrow ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)) \subseteq ((i_l \bowtie r) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$

so

$$j' = ((l \bowtie r) \uplus (((i_l \bowtie r) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- ((i_l \bowtie d_r) \uplus (i_r \bowtie d_l)))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

$$\{6\}$$

$$= ((l \bowtie r) \uplus ((((i_l \bowtie r) \uplus ((l \bowtie i_r) \uplus (i_l \bowtie i_r)))$$

$$- (i_l \bowtie d_r)) - (i_r \bowtie d_l))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

$\{4\}$

$$j' = ((l \bowtie r) \uplus (((((i_l \bowtie r) \uplus (l \bowtie i_r)) \uplus (i_l \bowtie i_r))$$
$$- (i_l \bowtie d_r)) - (i_r \bowtie d_l))) - (((r \bowtie d_l) \uplus (l \bowtie d_r)) - (d_r \bowtie d_l))$$

$\{5\}$

= expression A.2.5

$\{$substitution of j for $(l \bowtie r)\}$

□

A.3 Rewrite Rule Proof

In this appendix, a sample proof of one of the rewrite rules is presented. The following definitions are used in the proof.

- Definition A.3.1 *Equality of tuples*: Let tuple $\tau_s \in s(S)$ and tuple $\tau_t \in t(S)$. If $\forall \alpha \in S : \tau_s[\alpha] = \tau_t[\alpha]$ then $\tau_s = \tau_t$.

- Definition A.3.2 *Subtraction of tuples*: Let tuple $\tau_s \in s(S)$ and tuple $\tau_t \in t(S)$. If $\tau_s = \tau_t$ then $\tau_s - \tau_t = \emptyset$ else $\tau_s - \tau_t = \tau_s$.

- Definition A.3.3 *Join of tuples*: Let tuple $\tau_s \in s(S)$ and tuple $\tau_t \in t(T)$. Consider

a join operation $s \bowtie_{\mathsf{P}} t$. P is a predicate defined on a subset of $S \cup T$. If $\mathsf{P}(\tau_s, \tau_t)$ is true then $\tau_s\tau_t \in (s \bowtie_{\mathsf{P}} t)$ else $\tau_s\tau_t \notin (s \bowtie_{\mathsf{P}} t)$.

In the following proof we consider a distinct set of tuples $\tau_s \in s(S), \tau_t \in t(S)$, and $\tau_v \in v(V)$, and a join operation $\bowtie_{\mathsf{P}}$ where P is a predicate defined on $S \cup V$.

Rule 1: $(s - t) \bowtie v \longleftrightarrow (s \bowtie v) - (t \bowtie v)$

Proof:

The proof is presented in the form of truth table in Figure A.1. The first three columns of the truth table represent all the possible cases that we need to consider. The cases where predicate $\tau_s = \tau_t$ is true and $\mathsf{P}(\tau_s, \tau_v) \neq \mathsf{P}(\tau_t, \tau_v)$ are not shown. These states are not possible. If $\tau_s = \tau_t$ is true, then $\mathsf{P}(\tau_s, \tau_v) = \mathsf{P}(\tau_t, \tau_v)$ must be true. The value of the predicate in column 4, $\tau_s \in (s - t)$, is the inverse of predicate $\tau_s = \tau_t$. This follows from the definition of *equality of tuples* (Def. A.3.1) and the definition of the *subtraction of tuples* (Def. A.3.2). The value of the predicate in column 5, $\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v)$, is false if predicate $\mathsf{P}(\tau_s, \tau_v)$ in column 2 is false by the definition of the *join of tuples* (Def. A.3.3). The value of the predicate in column 5 is also false if $\tau_s \in (s - t)$ is false in column 4 since a tuple can not be joined if it doesn't exist. If $\mathsf{P}(\tau_s, \tau_v)$ is true and $\tau_s \in (s - t)$ is true, then $\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v)$ is true. The predicate $\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$ in column 6 is true if predicate $\mathsf{P}(\tau_s, \tau_v)$, column 2, is true, else it is false. This follows from the definition of the *join of tuples* (Def. A.3.3). The predicate $\tau_t\tau_v \in (t \bowtie_{\mathsf{P}} v)$ in column 7 is true if predicate $\mathsf{P}(\tau_t, \tau_v)$,

column 3, is true, else it is false. This also follows from the definition of the *join of tuples* (Def. A.3.3). Finally, in column 8, predicate $\tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$ is true if predicate $\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$ in column 6 is true and $\tau_t\tau_v \in (t \bowtie_{\mathsf{P}} v)$ in column 7 is false. It is also true if $\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$ in column 6 is true, and $\tau_t\tau_v \in (t \bowtie_{\mathsf{P}} v)$ is true, and $\tau_s = \tau_t$ in column 1 is false. In all other cases, predicate $\tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$ is false. This follows because if $\tau_s\tau_v \in (s \bowtie_{\mathsf{P}} v)$ is false, then $\tau_s\tau_v$ does not exist in the left operand of the subtraction, and will not be in the result of the subtraction. If $\tau_s\tau_v$ does exist in the left operand of the subtraction operation, then it can be removed if there exists an equivalent tuple in the right operand (Def. A.3.2). Otherwise it will not be removed and $\tau_s\tau_v$ will be included in the resulting relation of the subtraction operation.

By examining the table in Figure A.1 we see that either $\tau_s\tau_v \notin ((s - t) \bowtie_{\mathsf{P}} v) \wedge \tau_s\tau_v \notin ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$ is true or $\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v) \wedge \tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$ is true. In other words, column 5, $\tau_s\tau_v \in ((s - t) \bowtie_{\mathsf{P}} v)$, and column 8, $\tau_s\tau_v \in ((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$, match in every row. Therefore rule 1 holds. □

The proofs of the other rewrite rules are similar.

$\tau_s = \tau_t$	$\mathsf{P}(\tau_s, \tau_v)$	$\mathsf{P}(\tau_t, \tau_v)$	$\tau_s \in$ $(s - t)$	$\tau_s\tau_v \in$ $((s - t) \bowtie_{\mathsf{P}} v)$	$\tau_s\tau_v \in$ $(s \bowtie_{\mathsf{P}} v)$	$\tau_t\tau_v \in$ $(t \bowtie_{\mathsf{P}} v)$	$\tau_s\tau_v \in$ $((s \bowtie_{\mathsf{P}} v) - (t \bowtie_{\mathsf{P}} v))$
F	F	F	T	F	F	F	F
F	F	T	T	F	F	T	F
F	T	F	T	T	T	F	T
F	T	T	T	T	T	T	T
T	F	F	F	F	F	F	F
T	T	T	F	F	T	T	F

Table A.1: Truth table where T is TRUE and F is FALSE. The conditions $(\tau_s = \tau_t) = \mathsf{T}$ and $\mathsf{P}(\tau_s, \tau_v) \neq \mathsf{P}(\tau_t, \tau_v)$ are not shown since these states are not possible.

BIBLIOGRAPHY

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 245–256, San Jose, CA, May 1995.

[2] M. E. Adiba and B. G. Lindsay. Database snapshots. In *Sixth International Conference on Very Large Data Bases*, pages 86–91, Montreal, Quebec, Canada, October 1980.

[3] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems*, pages 175–186, Dallas, TX, May 2000.

[4] W. G. Aref and H. Samet. Efficient window block retrieval in quadtree-based spatial databases. *GeoInformatica*, 1(1):59–91, April 1997.

[5] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 747–756, New Orleans, LA, January 1997.

[6] J. Basch, L. J. Guibas, and L. Zhang. Proximity problems on moving points. In *Symposium on Computational Geometry*, pages 344–351, 1997.

[7] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 44–53, Edmonton, Canada, July 2002.

[8] T. Berners-Lee and D. Connolly. Hypertext markup language–2.0. Technical Report RFC 1866, Network Working Group, November 1995.

[9] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD Conference*, pages 61–71, Washington, D.C., June 1986.

[10] O. Buneman and E. Clemons. Efficiently monitoring relational databases. *ACM Transactions on Database Systems*, 4(3):368–382, September 1979.

[11] W. Cai, F.B.S. Lee, and L. Chen. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 82–89, Atlanta, GA, May 1999.

[12] C. S. Campbell and S. L. Egbert. Animated cartography: Thirty years of scratching the surface. *Cartographica*, 27(2):24–46, 1990.

[13] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 577–588, 1991.

[14] H. D. Chon, D. Agrawal, and A. E. Abbadi. Query processing for moving objects with space-time grid storage model. In *Third International Conference on Mobile Data Management*, pages 121–128, Singapore, January 2002.

[15] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the ACM SIGMOD Conference*, pages 469–480, Montreal, Quebec, Cananda, June 1996.

[16] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *Proceedings of the ACM SIGMOD Conference*, pages 405–416, Tucson, AZ, May 1997.

[17] R. Ding and X. Meng. A quadtree based dynamic attribute index structure and query process. In *2001 International Conference on Computer Networks and Mobile Computing Proceedings*, pages 446–451, Los Alamitos, CA, October 2001.

[18] D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[19] C. Esperança and H. Samet. Spatial database programming using SAND. In *Proceedings of the Seventh International Symposium on Spatial Data Handling*, volume 2, pages A29–A42, Delft, The Netherlands, August 1996.

[20] ESRI. Arcview tracking analyst. ESRI White Paper Series, 1998.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–http/1.1. Technical Report RFC 2616, The Internet Society, Network Working Group, June 1999.

[22] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[23] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD Conference*, pages 328–339, San Jose, CA, May 1995.

[24] L. J. Guibas, J. Hershberger, S. Suri, and L. Zhang. Kinetic connectivity for unit disks. In *Proceedings of the 16th ACM Symposium on Computational Geometry*, pages 331–340, 2000.

[25] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference*, pages 157–166, Washington, D.C., May 1993.

[26] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[27] E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM SIGMOD Conference*, pages 440–453, San Francisco, CA, May 1987.

[28] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, Zurich, Switzerland, September 1995.

[29] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[30] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. (Also University of Maryland Computer Science TR–3919).

[31] G. Iwerks and H. Samet. The spatial spreadsheet. In *Visual Information and information Systems: Third International Conference, VISUAL'99*, pages 317–324, Amsterdam, The Netherlands, June 1999. Springer-Verlag.

[32] G. Iwerks and H. Samet. Incremental view maintenance of spatial joins. Technical Report CS-TR-4179, University of Maryland, College Park, MD, August 2000.

[33] G. Iwerks and H. Samet. Visualization of dynamic spatial data and query results over time in a gis using animation. In *Visual Information and information Systems: Fourth International Conference, VISUAL'00*, Lyon, France, November 2000.

[34] G. S. Iwerks and H. Samet. The internet spatial spreadsheet: Enabling remote visualization of dynamic spatial data and ongoing query results over a network. In *ACM - GIS '03, Proceedings of the 11th international symposium on geographic information systems*, pages 154–160, New Orleans, LA, November 2003.

[35] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 512–523, Berlin, Germany, September 2003.

[36] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *To be published in Proceedings of the 30th International Conference on Very Large Data Bases*, Toronto, Canada, September 2004.

[37] B. Jobard and W. Lefer. The motion map: Efficient computation of steady flow animations. In *IEEE Symposium on Information Visualization, 1997, Proceed-*

ings, pages 323–328, Phoenix, AZ, October 1997.

[38] M. Kang and S. Servign. Animated cartography for urban soundscape information. In *Proceedings of the 7th Symposium on Geographic Information Systems*, pages 116–121, Kansas City, MO, November 1999. ACM.

[39] A. Koussoulakou and M. J. Kraak. Spatio-temporal maps and cartographic communication. *The Cartographic Journal*, 29:101–108, 1992.

[40] M. Kraak and A. M. MacEachren. Visualization of the temporal component of spatial data. In *Proceedings of Spatial Data Handling 1994*, pages 391–409, 1994.

[41] M. Levoy. Spreadsheets for images. In *Proceedings of the SIGGRAPH'94 Conference*, pages 139–146, Los Angeles, CA, July 1994.

[42] B. G. Lindsay, L. M. Haas, C. Mohan, H. Pirahesh, and P. F. Wilms. A snapshot differential refresh algorithm. In *Proceedings of the ACM SIGMOD Conference*, pages 53–60, Washington, D.C., May 1986.

[43] K. Ma, D. Smith, M. Shih, and H. Shen. Efficient encoding and rendering of time-varying volumn data. Technical Report NASA/CR-1998-208424 ICASE Report No. 98-22, National Aeronautics and Space Administration, Langley Research Center, Hampton. VA, June 1998.

[44] A. M. MacEachren, F. P. Boscoe, D. Haug, and L. W. Pickle. Geographic visualization: Designing manipulable maps for exploring temporally varying geo-

referenced statistics. In *IEEE Symposium on Information Visualization, 1998, Proceedings*, pages 87–94,156, Research Triangle Park, NC, October 1998.

[45] A. M. MacEachren and D. DiBiase. Animated maps of aggregate data: Conceptual and pratical problems. *Cartography and Geographic Information Systems*, 18(4):221–229, 1991.

[46] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[47] R. E. Meisner, M. Bittner, and S.W. Dech. Visualization of satellite derived time-series datasets using computer graphics and computer animation. In *1997 IEEE International Geoscience and Remote Sensing, 1997. IGARSS '97. Remote Sensing - A Scientific Vision for Sustainable Development*, pages 1495–1498, Oberpfaffenhofen, Germany, August 1997. IEEE.

[48] R. E. Meisner, M. Bittner, and S.W. Dech. Computer animation of remote sensing-based time series data sets. In *IEEE Transactions on Geoscience and Remote Sensing*, pages 1100–1106, Oberpfaffenhofen, Germany, March 1999. IEEE.

[49] H. Mokhtar, J. Su, and O. Ibarra. On moving object queries. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems*, pages 188–198, Madison, WI, June 2002.

[50] M. A. Nascimento, R. Silva, and Y. Theodoridis. Evaluation of access structures for discretely moving points. In *Proceedings of the International Workshop*

on Spatio-Temporal Database Management, pages 171–188, Edinburgh, UK, September 1999.

[51] Standards Committee on Interactive Simulation (SCIS). *IEEE Std 1278.1-1995*. IEEE Computer Society, USA, March 1996.

[52] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. 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.

[53] K. Raptopoulou, A. N. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica*, 7(2):113–137, 2003.

[54] J. Rasure and C. Williams. An integrated visual language and software development environment. *Journal of Visual Languages and Computing*, 2(3):217–246, September 1991.

[55] N. Roussopoulos and H. Kang. Principles and techniques in the design of adms±. *IEEE Computer*, 19:19–25, 1986.

[56] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, pages 71–79, San Jose, CA, May 1995.

[57] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proceedings. 18th International Conference on Data Engineering*, pages 463–472, San Jose, CA, February 2002.

[58] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the ACM SIGMOD Conference*, pages 331–342, Dallas, TX, May 2000.

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

[60] F. Schroder. Visualizing meteorological data for a lay audience. *IEEE Computer Graphics and Applications*, 13(2):12–14, September 1993.

[61] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, New York, NY, third edition, 1996.

[62] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of the 13th IEEE Conference on Data Engineering (ICDE)*, pages 422–432, Birmingham, U.K., April 1997.

[63] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the ACM SIGMOD Conference*, pages 65–78, San Jose, CA, May 1975.

[64] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *Proceedings of the ACM SIGMOD Conference*, pages 334–345, Madison, WI, June 2002.

[65] Y. Tao and D. Papadias. Spatial queries in dynamic environments. *ACM Transactions on Databases Systems (TODS)*, 28(2):101–139, June 2003.

[66] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[67] J. Woodfill and M. Stonebraker. An implementation of hypothetical relations. In *9th International Conference on Very Large Data Bases*, pages 157–166, Florence, Italy, October 1983.