TOWARDS A UNIFIED FRAMEWORK FOR EFFICIENT ACCESS METHODS AND QUERY OPERATIONS IN SPATIO-TEMPORAL DATABASES

by

Yun Chen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:

Associate Professor Jignesh M. Patel, Chair
Professor Timothy J. Gordon
Professor Hosagrahar V. Jagadish
Associate Professor Sugih Jamin

© Yun Chen 2008
All rights reserved.

DEDICATION

This thesis is dedicated to my father, who devoted his entire life to aiding my pursuit of knowledge and education. This thesis is also dedicated to my mother, who set a larger than life example of diligence and prudence. This thesis is also dedicated to my dear sister, without whose help none of this would have been possible. Finally, this thesis is dedicated to my beloved wife, who has always stood by me and given me unconditioned love and support.

ACKNOWLEDGEMENTS

Dr. Jignesh M. Patel has been the ideal thesis supervisor. His sage advice, insightful criticisms, out-of-the-box thinking, and patient encouragement aided the writing of this thesis in innumerable ways.

TABLE OF CONTENTS

DEDICATION . . . . **ii**

ACKNOWLEDGEMENTS . . . . **iii**

LIST OF TABLES . . . . **vii**

LIST OF FIGURES . . . . **viii**

CHAPTER

1. INTRODUCTION 1

2. EFFICIENT INDEXING OF PREDICTED TRAJECTORIES WITH STRIPES 6

2.1 Introduction 6

2.2 Background and Model 9

2.2.1 Query Types 10

2.3 TPR-Tree and TPR*-Tree Indices 11

2.3.1 TPR-Tree 11

2.3.2 TPR*-Tree 12

2.4 STRIPES 14

2.4.1 Dual Transform for Moving Objects 14

2.4.2 Index Structure 16

2.4.3 Insertion 17

2.4.4 Deletion 18

2.4.5 Update 19

2.4.6 Queries 19

2.5 Experimental Evaluation 23

2.5.1 Implementation Details and Experimental Platform . . . 23

2.5.2 Data Sets and Workload 25

2.5.3 Effect of Workload Mix 26

2.5.4 Scaling with Increasing Number of Moving Objects . . 29

2.5.5 Effect of Data Skew 31

2.5.6 Summary 31

2.6 Related Work 32

2.7 Conclusions 34

3. ALL NEAREST NEIGHBOR QUERY ALGORITHMS AND METRICS . . . 36
3.1 Introduction . . . 36
3.2 Related Work . . . 39
3.3 ANN Evaluation . . . 41
3.3.1 A New Pruning Distance Metric . . . 42
3.3.2 The ANN Algorithms . . . 51
3.3.3 Extension to AkNN . . . 57
3.3.4 MBRQT . . . 59
3.4 Experimental Evaluation . . . 60
3.4.1 Implementation Details . . . 60
3.4.2 Experimental Datasets and Workload . . . 62
3.4.3 Evaluating the ANN Algorithms . . . 63
3.4.4 Effectiveness of the NXNDIST Metric . . . 65
3.4.5 Comparison of BNN, MBA, and GORDER . . . 66
3.4.6 Effect of Dimensionality . . . 68
3.4.7 Evaluating AkNN Performace . . . 69
3.5 Conclusions . . . 70
4. TRAJECTORY JOINS WITH APPLICATION IN PRIVACY PRESERVATION . . . 72
4.1 Introduction . . . 72
4.2 Related Work . . . 75
4.2.1 Spatio-Temporal Joins . . . 75
4.2.2 Location Privacy . . . 76
4.3 Trajectory Join Operations . . . 76
4.3.1 Trajectory Distance Join (TDJ) . . . 77
4.3.2 Trajectory kNN Join (TkNNJ) . . . 79
4.3.3 Relaxations and Restrictions . . . 80
4.4 JiST Join Algorithms . . . 81
4.4.1 Representing Trajectories in JiST . . . 81
4.4.2 Time-Parameterized Distance Between Trajectories . . . 82
4.4.3 Naïve Algorithms . . . 83
4.4.4 Dual Index Based Algorithms . . . 88
4.5 Using Indices in JiST . . . 95
4.6 Trajectory Privacy Preservation . . . 97
4.6.1 Trajectory k-Anonymity . . . 97
4.6.2 Trajectory Privacy Policies . . . 99
4.6.3 Trajectory Cloaking with JiST . . . 100
4.6.4 Presenting *k*-Anonymous Trajectories . . . 105
4.7 Experimental Evaluations . . . 107
4.7.1 Implementation Details . . . 107
4.7.2 Experimental Settings . . . 108
4.7.3 Comparing with Naïve Algorithms . . . 109
4.7.4 Evaluating Dual Index Based Algorithms . . . 109
4.8 Conclusions and Future Work . . . 116

5. CONCLUSIONS 120

BIBLIOGRAPHY **122**

LIST OF TABLES

Table

2.1 Table of Frequently Used Notations 14

3.1 Table of Frequently Used Notations 42
3.2 Table of Experimental Datasets 62

4.1 Table of Frequently Used Notations 77
4.2 Experiment Parameters 107

LIST OF FIGURES

Figure

2.1	Query Examples for Objects Moving in a 1-d Space	11
2.2	Area Computation in the TPR-Tree	11
2.3	Motivation for the Insert Algorithm in TPR*-tree	11
2.4	Transformed One-dimensional Time-slice Query: Q1 from Figure 2.1	20
2.5	Transformed One-dimensional Window Query: Q2 from Figure 2.1	20
2.6	Transformed One-dimensional Moving Query: Q3 from Figure 2.1	20
2.7	Algorithm to Test the Relative Positions of Data and Query Regions	23
2.8	Relative Positions of Data and Query Regions for One-dimensional Points	23
2.9	500K-Uniform: Continuous Performance Measurement for Batch of 5K Operations	27
2.10	500K-Uniform: I/O and CPU Costs for 50K Operations	27
2.11	500K-Uniform: Average Single Update Costs	28
2.12	500K-Uniform: Average Single Query Costs	28
2.13	Effect of Number of Moving Objects	30
2.14	500K-Skewed: Single Query Costs	30
3.1	2-D Intuition of NXNDIST	45
3.2	3-D Computation of NXNDIST	45
3.3	Metrics on MBRs	45
3.4	Counter-Example for MAXMIN	47
3.5	Counter-Example for NXNDIST	47
3.6	Computation of 1-D MAXMIN	47
3.7	Evaluating the ANN Algorithms	63
3.8	Comparison of Index Structures and Methods: TAC Data(2D)	66
3.9	Comparison of Index Structures and Methods: FC Data(10D)	66
3.10	Effect of Dimensionality	69
3.11	AkNN on TAC Data(2D)	70
3.12	AkNN on FC Data(10D)	70
4.1	Trajectory Joins: A One-dimensional Example	73
4.2	Finding $TPD^{\vdash}(r, s, w'')$ and $TPD^{\dashv}(r, s, w'')$	83
4.3	The JiST Time Partitioned Storage Model	87
4.4	The General JiST Index Structure	89
4.5	TBR Construction and Distance Metrics	90

4.6 Trajectory k-Anonymity with TkNNJ 98
4.7 Trajectory k-Anonymity with TDJ 105
4.8 Presenting k-Anonymous Trajectories 106
4.9 Comparing with Naïve Algorithms: Historical 109
4.10 TDJ−Effect of Join Distance Δd 110
4.11 TkNNJ−Effect of k 111
4.12 Effect of Query Window Size w: TDJ 112
4.13 Effect of Query Window Size w: TkNNJ 112
4.14 Effect of Workload Mix: TDJ 113
4.15 Effect of Workload Mix: TkNNJ 115

CHAPTER 1

INTRODUCTION

The need to accurately determine location has been a critical part of human exploration endeavors since the early days of human history. The advent of satellite and atomic clock technologies in the second half of the last century has led to the development of satellite based location detection techniques such as the GPS [36]. For a long time, the GPS technology was only available for military applications, but since May of 2000 it has become possible for *civilian* GPS receivers to accurately identify the location of a GPS receiver within a few meters. GPS technology is quickly enabling numerous new Location Based Service (LBS) applications, including tracking fleets of vehicles, navigating ships and aeroplanes, and tracking wildlife. Another popular application of this technology is in cellular phones with embedded GPS sensors. In the United States, cellular phones are now required to be E911 [30] enabled. E911 is a federally mandated requirement which states that cellular phone companies must be able to locate the geographical location of the cellular phone user with an accuracy of a few hundred meters in most cases. In case of an emergency, E911 will provide better location information to the emergency workers. The reverse use of the E911 service (Reverse 911) [1] can also enable prompt delivery of automated warnings to civilians in case of a crisis. For example, Reverse 911 has been successfully deployed and put to use in the 2007 San Diego wildfire [4] and has helped over 500,000 families evacuate from the danger zone [5].

While GPS works well for location detection in the outdoors, determining location when inside a building requires using different techniques. A popular technique for

determining location indoors is using ultrasonic, or radio frequencies, or a combination of these two techniques [7, 45, 89, 110]. Location information produced by these techniques can be very accurate; for example, the BAT system [45, 110] can compute locations that is usually accurate within 10 cm of the actual location. The availability of indoor location information is also giving rise to a new class of applications, such as asset tracking and context-aware applications that adapt to the user's needs as a user moves around in physical space. Rapid advances in semiconductor technologies have made it possible to build such location-computing devices for a small price, making mass deployments of such devices possible.

These location-computing techniques allow us to gather successive location updates of an object, which can be either a user or a device, to produce a sequence of locations that collectively form the *trajectory* for that object. (A trajectory segment for an object moving in 2-D physical space is essentially a line in 3-D space, with time as one of the dimensions.) In the last few years, we have seen a rapid increase in the deployment of location-sensing devices and applications that make use of this information, and this trend is likely to accelerate in the near future. As a result, we will soon be faced with the task of managing large volumes of trajectory data. For example, if one were to continually collect GPS sensor readings from a fleet of trucks, then in a short amount of time one would have a large volume of trajectory data. Such trajectory data sets can be useful in safety research such as analysis of factors that contribute to accidents, including car behavior such as cruising speed, lane switching, following distance, etc., especially when combined with other data such as sensor readings from the braking system.

Spatio-temporal databases, also called moving object databases, are required to support queries on large numbers of continuously moving objects. A key requirement for indexing methods in this domain is to efficiently support both update and query operations. Previous work on indexing such databases can be broadly classified into two categories: indexing the past positions and indexing the future predicted positions. In Chapter 2 we focus on an

efficient indexing method for indexing the future predicted positions of moving objects.

In Chapter 2 we propose an indexing method called STRIPES that indexes predicted trajectories in a dual transformed space. Trajectories for objects in d-dimensional space are transformed into points in a $2d$-dimensional space. This dual transformed space is then indexed using a hierarchical grid decomposition index structure. STRIPES can evaluate a range of queries including time-slice, window, and moving queries. We perform extensive experimental evaluation comparing the performance of STRIPES with the leading existing predicted trajectory index (the TPR*-tree), and show that our approach is significantly faster than TPR*-tree for both updates and search queries.

The All Nearest Neighbor (ANN) operation is a commonly used primitive for analyzing large multi-dimensional spatial datasets. Since computing ANN is very expensive, in previous works R*-tree based methods have been proposed to speed up this computation. These traditional index based methods use a pruning metric called MAXMAXDIST, which allows the algorithms to prune nodes in the index that need not be examined during the ANN computation. In Chapter 3 we introduce a new pruning metric called NXNDIST, and show that this metric is far more effective than the traditional MAXMAXDIST metric. In addition, we also explore a range of algorithms for computing ANN, which differ in the way the spatial index is traversed and nodes are expanded. We show that one of these methods, which uses a depth-first index traversal and bi-directional node expansion, consistently outperforms all other methods. Furthermore, we show that our method can also efficiently evaluate the more general All-k-Nearest-Neighbor (AkNN) operation.

In Chapter 3, we also challenge the common practice of using R*-tree index for speeding up the ANN computation. We propose an enhanced bucket quadtree index structure, called MBRQT, and using extensive experimental evaluation show that the MBRQT index offers better speedup in ANN computation than the commonly used R*-tree index.

Traditional spatial and traditional temporal joins have been widely studied in the past,

but there is very little work on the more complex problem of trajectory join operations, which have many interesting emerging applications, privacy preservation in publishing trajectory data being one of them. As another example application, during the Mad Cow Disease epidemic, in the event where some animals are identified as having contracted the disease, a decision needs to be made quickly regarding which other animals have come close to the infected animals within a certain period of time in the past. This computation is essentially a trajectory join operation.

In Chapter 4 we present a general framework, called **JiST**, that introduces a broad class of trajectory join operations, including *Trajectory Distance Join (TDJ)* and *Trajectory k Nearest Neighbors Join (TkNNJ)*. Within the JiST framework, we present a set of algorithms to evaluate the join operations introduced in this chapter. We adapt the STRIPES index structure presented in Chapter 2 and propose a unified index structure that incorporates indexing both historical and future trajectories of moving objects. We then introduce the notions of *Trajectory k-Anonymity* and *Trajectory Cloaking*, and show the application of the JiST operations in the context of privacy preservation. Finally, we present results from detailed experiments that demonstrate the efficiency and scalability of the JiST join algorithms. To the best of our knowledge, JiST is the first comprehensive framework for complex trajectory join operations and paves the foundation for building a complex querying platform for emerging trajectory based applications.

This thesis makes the following contributions: we propose an efficient access method called STRIPES for indexing and querying predicted trajectories; we devise a new pruning metric and develop a number of algorithms for processing complex *ANN* queries in spatial databases, which can be employed in efficiently querying snapshots of spatio-temporal databases; we extend the STRIPES index structure to unify the indexing techniques of past, present, and future trajectories; we identify and propose algorithms for the set of complex spatio-temporal join operations that span the entire timeline of the past, the present and the future and form the foundation of the comprehensive JiST framework for

processing trajectory joins; we also formulate the intriguing concept of trajectory privacy and demonstrate the ease and flexibility with which the JiST framework can be applied to solve a new class of problems.

CHAPTER 2

EFFICIENT INDEXING OF PREDICTED TRAJECTORIES WITH STRIPES

2.1 Introduction

Over the last decade, we have witnessed an increasing interest in techniques for managing databases consisting of a large number of continuously moving objects. These research interests have been fuelled by rapid advances in hardware technologies that allow for cheap location-aware devices, which are often packaged in small physical devices. These devices have found applications in a variety of civilian and military monitoring applications. Many of these applications demand extremely efficient techniques for dealing with a large update rate triggered by object continuously updating their location information. In addition, these application also require efficient techniques for querying on the location information. Queries in these applications can be divided into two broad classes: queries on the past positions of moving objects, and queries on predicted positions of the moving objects. In this chapter, we focus on this later class of queries, which are often referred to as predictive queries. While there are a number of proposals for modeling the predicted positions of moving objects, the most widely-used model specifies the predicted position as a current position and a velocity vector indicating the direction of the future motion [43, 59, 72, 81, 85, 86]. We also use this commonly-used model.

To efficiently answer queries on predicted position, it is intuitive to ask the question if effective indexing methods can be built for these moving object data sets. Naturally, a substantial amount of research has been undertaken in the recent past to answer

this question. Some of the early work in this area employs dual transformation techniques [6, 59, 111]. These techniques typically represent the predicted position of an object moving in a d-dimensional space as a point in a $2d$-dimensional space. Most of this body of work is largely theoretical in nature, and for most parts focuses on objects moving in a one-dimensional space [59]. More recent work in this area has focused primarily on practical implementations of indexing structures for predictive queries [21, 70, 72, 87, 88, 90, 105]. Of these indexing methods, perhaps the most influential indexing method is the TPR-tree [72]. This indexing structure uses the basic R*-tree indexing structure [10], and expands the traditional definition of bounding boxes to include time-parameterized bounding boxes. Essentially each bounding box now has an associated velocity vector that captures the growth of the box as time progresses. The TPR-tree has inspired a flurry of research aimed at improving the basic TPR-tree algorithms. A recently proposed indexing structure, called the TPR*-tree [105] has been shown to vastly outperform the basic TPR-tree index.

Surprisingly, in the more empirical research on predictive trajectory indexing, the early dual transformed methods have been largely dismissed. (The TPR-tree [72] employs techniques that are inspired by the dual transformed methods, but doesn't actually experimentally compare the TPR-tree index with any dual transform based methods.) Perhaps a reason for this dismissal is because a) the research in dual transform indexing methods has largely focused on deriving asymptotic performance bounds, and b) these researchers have suggested that the dual transformed space be indexed using methods such as partitions trees [59], which are not as widely used as R-trees in practice. The body of empirical research has largely dismissed these theoretical results arguing that the asymptotic performance bounds, though interesting, don't lead to practical structures since these bounds have large constant factors [72, 105].

In this chapter, we reexamine this issue and consider if a practical indexing structure can be built using a dual transformation technique. The motivation for our interest in

this approach stems from the observation that R-tree based indexing techniques are known to rapidly degrade in performance as the dimensionality of the underlying space increases [12]. All the TPR-tree based indexing structures use time-parameterized boxes, which require representing a velocity value for each dimension. In effect, the underlying indexing space can be viewed as a $2d$-dimensional space for objects moving in a d-dimensional space. The dual transform techniques also need to index in a $2d$-dimensional space. However, rather than indexing boxes the dual transform techniques only have to index points, which is potentially easier to index efficiently. This key insight is at the heart of the STRIPES indexing structure, which we propose in this chapter.

STRIPES is a Scalable Trajectory Index for Predicted Positions in Moving Object Databases. The STRIPES index maps predicted positions to points in a dual transformed space and indexes this space using a disjoint regular partitioning of space. This style of partition is called quadtrees, and can be extremely efficient, especially for indexing point data [93]. Even though the traditional quadtree leads to an unbalanced tree, it has proven to be an effective disk based indexing structure in some cases [33,47]. STRIPES essentially employs a disk based bucket PR quadtree structure [93]. STRIPES can evaluate the entire range of predictive queries, which include time-slice, window, and moving queries [72].

In this chapter, we compare the performance of STRIPES with the currently best known indexing structure, namely the TPR*-tree [105]. Using actual implementations of these two indices on top of the SHORE storage manger [22], we demonstrate that STRIPES outperforms the TPR*-tree for both updates and query operations. **In most cases, updates in STRIPES are more than an order of magnitude faster that the TPR*-tree, and queries are about 4x faster with STRIPES!**

This chapter makes an important contribution which includes proposing a new indexing structure that is extremely efficient for predictive queries. In addition, we have essentially come to a full circle with this work, where we now show that the intuition behind the earlier theoretical work on dual transform based techniques can indeed be leveraged to

produce a practical and efficient predicted trajectory indexing method.

The remainder of this chapter is organized as follows: In Section 2.2, we cover the model used for representing predicted positions. Section 2.3 describes the TPR-tree and the TPR*-tree indices. The STRIPES index is described in Section 2.4, with experimental results in Section 2.5. Related work is reviewed in Section 2.6, and we present our conclusions and plans for future work in Section 2.7.

2.2 Background and Model

Location data for moving objects is continuously changing between any two successive updates of the location of the mobile object. This poses a problem in representing the location of the object at all times because most conventional models for data representations are static in nature. A commonly used model for representing trajectory data approximates the motion of an object as a straight line segment between two consecutive updates [43, 59, 72, 81, 85, 86]. The same linear model is used for predicting future trajectories as well [72, 105]. The object is assumed to move with some specified current velocity from the current position until a new update is explicitly issued. If the current position and velocity of an object is represented as $(\overline{p}(t), \overline{v}(t))$ at time t, then the position at time $t'(t' > t)$ can be calculated using $\overline{p}(t) = \overline{p}(t') + \overline{v}(t) \times (t - t')$. When the actual update arrives, which can be different from the predicted position, the velocity and the current positions are updated in the index to reflect the new predicted trajectory.

As has been noted in previous studies, when indexing predicted trajectories an optimal packing of a group of objects into a node in the index at time t is unlikely to be optimal at a later time t'. Consequently, an index that is optimal for queries at t will not be optimal at time t', and the index performance gradually deteriorates as t' increases. To reduce the dramatic performance degradation of an index built at a long time in the past, the trajectory indexing mechanisms often employ the notion of an index lifetime [72]. The lifetime is the time interval for which the index is designed to give good performance. After this

time interval the performance of the index is likely to deteriorate. The index is rebuilt periodically to avoid such rapid deterioration.

Another practical observation is that for an object moving in a d-dimensional space, the predicted trajectory includes a d-dimensional current spatial coordinate, and a d-dimensional velocity vector. Consequently indexing predicted trajectories requires indexing these two d-dimensional entities, which essentially requires indexing entities in $2d$-dimensional space. The so called curse of dimensionality [11], and the related challenges with query evaluation methods in high-dimensional space [12, 14] quickly start becoming performance issues in this domain. In addition, since the optimal node for a new update can be different from the node containing the old representation for the object, updating the predicted trajectory of an index will often result in traversing multiple paths down an index.

2.2.1 Query Types

There are three classes of future queries that have been extensively used in the previous research for querying on predicted trajectories [72]. These three classes are time-slice query, window query, and moving query. For a one-dimensional space, Figure 2.1 shows one example for each of these query types. In this figure the x-axis represents the time dimension, and the y-axis represents the single spatial dimension.

In Figure 2.1, $Q1$ is a time-slice query, which finds all objects at some specified future time t in some spatial region R. $Q2$ is a window query for finding all objects in time window $[t, t']$ in region R. $Q3$ is a moving query to find all objects in time window $[t, t']$ in region R that is moving with velocity v.

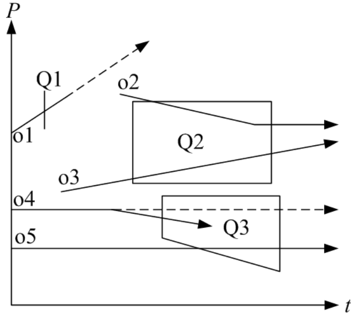


Figure 2.1: Query Examples for Objects Moving in a 1-d Space

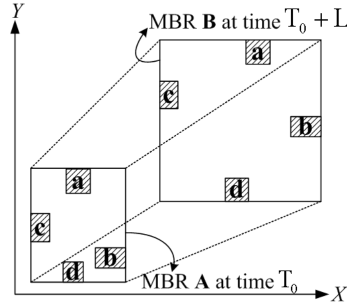


Figure 2.2: Area Computation in the TPR-Tree

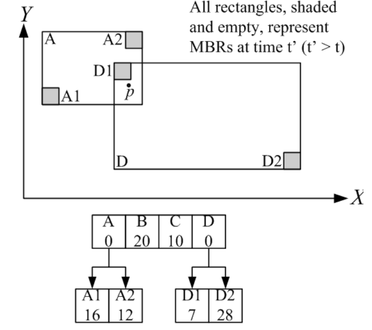


Figure 2.3: Motivation for the Insert Algorithm in TPR*-tree

2.3 TPR-Tree and TPR*-Tree Indices

In this section we review the two popular predicted indices, TPR-tree index [72] and the TPR*-tree index [105].

2.3.1 TPR-Tree

The TPR-tree [72] is essentially a time parameterized R*-Tree. The index stores velocities of the elements along with their positions in nodes. Since the elements are not static, the corresponding MBRs are dynamic (see Figure 2.2).

The index structure as well as the algorithms for search, insert and delete used are very similar to that of R*-tree [10]. The R*-tree uses a number of static parameters such as the area, perimeter, distance from the centroid, and the intersection between the two MBRs. The TPR-tree uses time parameterized metrics for these parameters. The time parameterized metric is computed using the formulae $\int_{T_0}^{T_0+L} M(t)dt$, where $M(t)$ is some metric that is used in the original R*-tree (for example the area), and L is the *lifetime* (see Section 2.3) of the index. The lifetime of an index is the time for which the index is used and queried.

Figure 2.2 shows an example of the time-parameterized area metric for four objects a, b, c, and d, moving in a two-dimensional space. In this figure, the actual data objects are shown as shaded boxes. The MBR of the index node at time T_0 is labeled as A, and

the MBR of the node at time $T_0 + L$ is labeled as B. The size and the position of B are calculated by extrapolating (position, velocity) of the entries with in the node. Then the area metric used is the volume of the trapezoid that is formed by the moving MBR of the node from time T_0 to $T_0 + L$. All the other metrics are computed in a similar fashion.

The insert algorithm chooses a node such that the expansion in volume is the smallest at non-leaf nodes and the expansion in integrated perimeter is the smallest at the leaf node level. When such a node is full, it is split similar to R*-Tree. Now instead of just sorting boundaries of elements, the velocity vectors are also sorted to choose the best distribution of the elements.

The TPR-Tree inherits all the problems related to the R*-Tree, such as overlap and dead space. Since the positions and the velocities are estimated and can change, the optimal combination of elements can not be maintained at all times in the future.

2.3.2 TPR*-Tree

The recently proposed TPR*-tree [105] provides a number of optimization over the basic TPR-tree algorithms. The key observation made by the authors is that during an insertion operation making a choice based on a local optimization may lead to a poor performing predicted trajectory index. To illustrate this key insight, consider the example shown in Figure 2.3. This figure shows a number of MBRs in the TPR-tree at a given time for objects moving in a two-dimensional space. In this example, the point p is being inserted into the index.

In an R-tree based insert algorithm, a least deterioration cost node is chosen for inserting the point p. In Figure 2.3, at the top level, the least deterioration cost is for nodes A and D. The greedy algorithm in TPR-Tree will pick node A because it requires the least area and perimeter expansion. And so at the level 2, node $A2$ will be picked for inserting the point p. However, the overall optimal node is for this insertion is node $D1$, which is the descendant of node D. The insert algorithm of TPR*-tree recognizes that

a local optimal solution at a level (node A in the example shown in Figure 2.3) can be from a broken-tie resulting from two elements having the same deterioration, and that the sub-elements (node $A2$ in this example) of that element may not be optimal. It proposes a novel *ChoosePath* algorithm that determines the node at any level that has the least cost of deterioration. It maintains a priority queue that is ordered by the cost of deterioration for each node, and the node with the least cost is picked for traversal at each step of the algorithm. The traversal continues until the bottom-most non-leaf node that has the least cost is found. This node is then chosen as the candidate for insertion. The authors argue that the extra cost incurred in traversing can be offset by the benefits of finding an optimal node for insertion. This algorithm leads to a tighter packing of elements in nodes and thus better query and insert performance.

The algorithm to deal with overflow nodes in TPR*-Tree is to first force reinsert and then split the node. For objects moving in a two dimensional space, the nodes are first sorted along all the eight ($4 \times d$) possible dimensions and the first $\lambda (= 30\%)$ entries from the best possible sort are chosen for reinsert. If during the reinsert, a node overflows then the node is split. The authors propose a heuristic to reduce the number of sorts to just one, by recognizing that the elements at leaf nodes can be assumed to be uniformly distributed, and the largest extent of all the dimensions (positions and velocities) would give the best benefit.

The TPR*-tree authors also propose a cost model, and a hypothetical optimal tree for predictive indices using a TPR-tree style of indexing. They show that the performance of the TPR*-tree is very close to the optimal index. *Consequently, one can conclude that the TPR*-tree is currently the best known practical indexing technique for predicted trajectories*.

Table 2.1: Table of Frequently Used Notations

Notation	Description
d	Number of dimensions in real space.
D	Number of dimensions in dual transformed space ($D = 2d$).
L	Index lifetime.
t_{ref}	Reference time, a.k.a. initialization time of an index.
$v_{max\mathtt{i}}$	Maximum absolute velocity value in i^{th} dimension.
$\overline{v}_{max}$	Vector of maximum velocities.
$p_{max\mathtt{i}}$	Maximum position value of a moving object in the ith dimension. In the ith dimension the position of an object ranges between 0 and $p_{max\mathtt{i}}$.
$\overline{p}_{max}$	Vector defining the dimensions of the physical space.
p_i	Position of an object in the dimension in original space.
$\overline{p}$	Position vector of an object in original space.
v_i	Velocity of an object in the dimension of original space.
$\overline{v}$	Velocity vector of an object in original space.
$p_{ref\mathtt{i}}$	Reference position of an object at time t_{ref} in i^{th} dimension of original space.
$\overline{p}_{ref}$	Reference position vector of an object in original space at time t_{ref}.
$P_{ref\mathtt{i}}$	Reference position of an object in i^{th} plane of transformed dual space.
$\overline{P}_{ref}$	Reference position vector of an object in dual transformed space.
$\overline{P}$	Position vector of an object in dual transformed space.
$\overline{V}$	Velocity vector of an object in dual transformed space.
$P_{ref\mathtt{i}}$	Reference position of an object in i^{th} plane of transformed dual space.
f	Non-leaf node fanout, $f = 2^D$.

2.4 STRIPES

In this section, we introduce the STRIPES index. To facilitate our discussion, we will use the notations described in Table 2.1.

2.4.1 Dual Transform for Moving Objects

The STRIPES index represents the moving object in a dual transformed space. The basic idea of a dual transform technique for predictive queries [6, 59] is to transform a linear trajectory defined by equation $\overline{p} = \overline{p}_{ref} + \overline{v}\left(t - t_{ref}\right)$ in $(d + 1)$-dimensional space (t being the additional dimension) into a point $\left(\overline{V}, \overline{P}_{ref}\right)$ in $2d$-dimensional dual space. Here, $\overline{V} = (V_1, V_2, \cdots, V_d)$, and $\overline{P}_{ref} = \left(P_{ref1}, P_{ref2}, \cdots, P_{ref\mathtt{d}}\right)$ are the transformed velocity and reference position vectors. We incorporate both negative and positive values for velocity

by applying the following transform: Given $(\overline{v}, \overline{p})$, the velocity and position vectors of an object, the corresponding transformed velocity and reference position vectors, $\left(\overline{V}, \overline{P}_{ref}\right)$ are calculated as follows:

$$\overline{V} = \overline{v} + \overline{v}_{max}$$

$$\overline{P}_{ref} = \overline{p} - \left(\overline{V} - \overline{v}_{max}\right)\left(t - t_{ref}\right)$$

Thus the range for $\overline{V}$ is $\left[\overline{0},\ 2\overline{v}_{max}\right]$, and the range for $\overline{P}_{ref}$ is

$$\left[-\overline{v}_{max} \times \left(t - t_{ref}\right),\ \overline{p}_{max} + \overline{v}_{max} \times \left(t - t_{ref}\right)\right].$$

Since time is monotonically increasing, the value of $P_{ref\mathtt{i}}$ is not bounded, which makes building an index that extends into infinity impossible. To solve this problem, we use the same technique that has been used in previous works [59, 72, 105], namely requiring that objects periodically issue an update to maintain a valid entry in the index. This time period is essentially the lifetime L of the index. As in previous works [59, 72, 105], we also employ a two-index strategy where we keep two distinct index structures in the system. The first index covers the time range from 0 to L, and the second index covers the time range from L to $2L$. The reference time of the first index is $t_{ref1} = 0$ and the reference time for the second index is $t_{ref2} = L$. Since an update consists of the deletion of the old entry and the insertion of the new entry, when an update with timestamp $> 2L$ arrives, we can simply delete the entries in the first index (either it is empty or the entries in that index have *expired* their lifetime [59, 72, 105]). At this point, we clear the first index structure and update its t_{ref} to $2L$. New updates with timestamps in the range are now inserted into this index. Using this strategy, we can observe that the range for $\overline{P}_{ref}$ in each of the indexes is $[-\overline{v}_{max} \times L,\ \overline{p}_{max} + \overline{v}_{max} \times L]$. To simplify the computation of index entry coordinates, we add $\overline{v}_{max} \times L$ to $\overline{p}_{ref}$ at transform time, and convert the range to $\left[\overline{0}, \overline{p}_{max} + 2 \times \overline{v}_{max} \times L\right]$. Thus, the transform equation becomes:

$$\overline{p}_{ref} = \overline{p} - (\overline{v} - \overline{v}_{max})\left(t - t_{ref}\right) + \overline{v}_{max} \times L$$

And, the linear motion equation becomes:

$$\overline{p} = \overline{p}_{ref} + (\overline{v} - \overline{v}_{max})\left(t - t_{ref}\right) - \overline{v}_{max} \times L.$$

2.4.2 Index Structure

The STRIPES index is essentially a disk based multidimensional PR bucket quadtree. Each of the d dual planes, $(V_1, P_1), (V_2, P_2), \cdots, (V_d, P_d)$, are equally partitioned into four *quads*. This partitioning results in a total of $4^d = 2^{2d}$ partitions, which we call *grids*. The fanout of non-leaf nodes f is thus 2^{2d}.

A non-leaf node stores the following information:

1. *level*: indicating the level of the non-leaf node.

2. *grid*: which encodes information about the quadrant corresponding to this node, in each of the d dual planes. In our implementation we simply indicate the quadrant by the lower vertex of the quadrant (this increases storage cost, but reduces computation time).

3. *children pointer array*: an array of 2^{2d} children pointers

4. *isLeaf array*: a vector of length 2^{2d} indicating whether each of the 2^{2d} children pointers point to a leaf or a non-leaf node.

5. *size*: indicating total number of actual data entries stored in all the leaf nodes in the subtree below this non-leaf node.

Leaf nodes store the *level*, *grid*, and *size* information, and the set of points that are being stored in the leaf node.

We note that the grids consist of a series of d quads from the d two-dimensional planes (i.e. the planes (V_1, P_1), (V_2, P_2), $\cdots$, (V_d, P_d)). Thus each grid is uniquely defined by the tuple $(\overline{V}, \overline{P}_{ref}, \overline{SL_V}, \overline{SL_P})$, where $\overline{V}\left(\overline{P}_{ref}\right)$ is the vector of velocity (reference position) coordinates of the leftmost (lowest) vertex of the d quads, and $\overline{SL_V}\left(\overline{SL_P}\right)$ is the vector of side lengths along the velocity (reference position) axis of the d quads.

2.4.3 Insertion

Being a dynamic index structure, STRIPES allows the insertion of objects on the fly.

We first discuss the algorithm used to find the target leaf node given an object to insert into the index.

Given the tuple $(\overline{v}, \overline{p})$ of a moving object, we first obtain transformed $\left(\overline{V}, \overline{P}_{ref}\right)$ tuple using the transform algorithm discussed in section 2.4.1. Then starting from the root node, we recursively identify the next level target node by calculating its array index in the *children pointer array* using the following formula:

$$\begin{aligned} array\ index &= \sum_{D=1}^{d} 2^{2D-1}\left(\left\lceil \frac{P_{refD}-P'_{refD}}{SL'_{PD}/2} \right\rceil - 1\right) \\ &+ \sum_{D=1}^{d} 2^{2D-2}\left(\left\lceil \frac{V_D - V'_D}{SL'_{VD}/2} \right\rceil - 1\right) \end{aligned} \tag{2.1}$$

where V'_D, P'_{refD}, SL'_{VD}, and SL'_{PD} are the velocity, reference position, velocity side length, and reference position side length parameters of the current node in the Dth dual plane.

The recursion terminates when either of the following two cases occurs: i) the target leaf node is non-existent; ii) the target leaf node is found. Since for case ii) there are two sub-cases considering whether the leaf node is full or not, we end up having to consider the following three cases during an insert operation:

Case 1: the target leaf node is non-existent.

Case 2: the target leaf node is found and not full.

Case 3: the target leaf node is found and is full.

Next, we discuss each of these cases in turn. In case 1, a new leaf node is created, and the new entry is inserted into this node. The grid parameters for the new leaf node are determined as follows:

$$\begin{cases} \overline{SL}_V &= \overline{SL}'_V/2 \\ \overline{SL}_P &= \overline{SL}'_P/2 \\ \overline{P}_{Gref} &= \left(\left\lceil \frac{\overline{P}_{ref}}{\overline{SL}_P} \right\rceil - \overline{1}\right) \times \overline{SL}_P \\ \overline{V}_G &= \left(\left\lceil \frac{\overline{V}}{\overline{SL}_V} \right\rceil - \overline{1}\right) \times \overline{SL}_V \end{cases}$$

(Note: Multiplications and divisions between vectors in the above equation imply element-wise operations.)

In the above equations, $\left(\overline{V}_G, \overline{P}_{Gref}, \overline{SL}_V, \overline{SL}_P\right)$ are the grid parameters of the newly created leaf node; the vectors $\left(\overline{V},\ \overline{P}_{ref}\right)$ correspond to the new entry being inserted; $\overline{SL}'_V$ and $\overline{SL}'_P$ are existing grid parameters of the current node.

In case 2, the object is directly inserted into the leaf node.

In case 3, a split operation is performed, where the target leaf node is promoted to a non-leaf node, and new leaf nodes are created. For creating the new leaf nodes, we follow the same process as defined in case 1, and reinsert data entries from the old leaf node in the sub-tree below this new non-leaf node.

An important aspect of this indexing structure is that new nodes are created only when necessary, which result in an efficient insert operation. The drawback of this approach is that it results in an unbalanced tree. However the actual disk space used for the non-leaf nodes is small, and often can stay resident in the buffer pool.

2.4.4 Deletion

When the motion parameters of an object are updated, the delete method is invoked to remove the previous entry for the object. Objects send in updated motion parameters together with the old parameters which are used to locate their old entries in the index. The method for locating the old entry recursively applies Equation 1 (see Section 2.4.3) to locate the leaf node that contains this object. (Recall from the discussion in section 2.4.1 that it is possible that this object may have *expired*. In this case the update is simply treated as an insert for a new object.)

At deletion time, non-leaf nodes are checked for under-fill, which is defined as whether the number of objects contained in the subtree below this node (indicate by the size information stored in the non-leaf node) is less than or equal to the capacity of a leaf node. The following two cases apply:

Case 1: The non-leaf node is not under-filled, in which case the target entry is directly deleted.

Case 2: The non-leaf node is under-filled, in which case all the entries within this node are first collected. Then, this node is converted to a leaf node, and the collected entries are re-inserted into the new leaf node. Finally, the target entry is deleted.

2.4.5 Update

Updates issued by objects contain the tuple $((t_{old}, \overline{v}_{old}, \overline{p}_{old}), (t_{new}, \overline{v}_{new}, \overline{p}_{new}))$ and are evaluated as a delete followed by an insert. The t_{old} and t_{new} reference times are used to determine which of the two indexes the old and new entry belong to.

2.4.6 Queries

We consider three types of queries: time-slice query, window query, and moving query, which were originally defined in [72].

For ease of reference, we modify the definition in [72] to better fit within our context.

Let $\overline{p}_l < \overline{p}_u$, $\overline{p}_{1l} < \overline{p}_{1u}$, and $\overline{p}_{2l} < \overline{p}_{2u}$ be the vectors of lower bounds and upper bounds in position, and t, t_l, t_u be three time instants not earlier than current time, such that $t < t_u$, and $t_l < t_u$. The three types of queries can now be defined as:

Time-slice query: $Q = (\overline{p}_l, \overline{p}_u, t)$ specifies a hyper-rectangle bounded by $[\overline{p}_l, \overline{p}_u]$ at time t.

Window query: $Q = (\overline{p}_l, \overline{p}_u, t_l, t_u)$ specifies a hyper-rectan-gle bounded by $[\overline{p}_l, \overline{p}_u]$ that covers the time interval $[t_l, t_u]$, i.e., this query retrieves points with trajectories in $\overline{p} - t$ space crossing the $(d + 1)$-dimensional hyper-rectangle $([p_{1l}, p_{1u}], [p_{2l}, p_{2u}], \cdots, [p_{dl}, p_{du}], [t_l, t_u])$.

Moving query: $Q = ([\overline{p}_{1l}, \overline{p}_{1u}], [\overline{p}_{2l}, \overline{p}_{2u}], t_l, tu)$ specifies the $(d + 1)$-dimensional trapezoid obtained by connecting the hyper-rectangle bounded by $[\overline{p}_{1l}, \overline{p}_{1u}]$ at time t_l and the hyper-rectangle bounded by $[\overline{p}_{2l}, \overline{p}_{2u}]$ at time t_u.

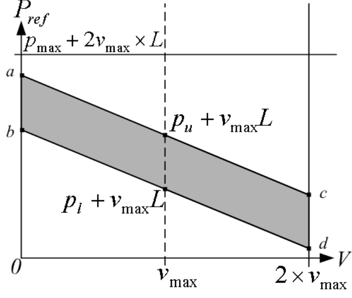


Figure 2.4: Transformed One-dimensional Time-slice Query: Q1 from Figure 2.1

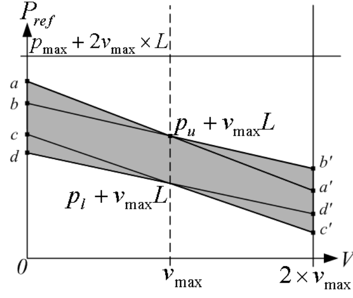


Figure 2.5: Transformed One-dimensional Window Query: Q2 from Figure 2.1

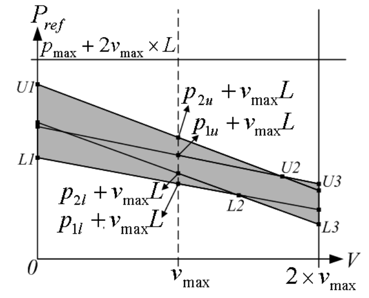


Figure 2.6: Transformed One-dimensional Moving Query: Q3 from Figure 2.1

Figure 2.1 illustrates the three query types on objects moving in a native one-dimensional space.

In Figure 2.1, $Q1$ is a time-slice query that returns object $o1$, $Q2$ is a window query that returns objects $o2$ and $o3$, and $Q3$ is a moving query that returns objects $o4$ and $o5$.

The most general query type is the moving query, which is of the form $Q = ([\overline{p}_{1l}, \overline{p}_{1u}], [\overline{p}_{2l}, \overline{p}_{2u}], t_l, t_u)$. Window queries are essentially moving queries with $\overline{p}_{1l} = \overline{p}_{2l}$, and $\overline{p}_{1u} = \overline{p}_{2u}$, whereas time-slice queries are just window queries with $t_l = t_u$. In essence, the general query Q translates into the following set of inequalities:

$$\begin{cases} \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L \geq \overline{p}_{1l} \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L \leq \overline{p}_{1u} \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_u - t_{ref}\right) - \overline{v}_{max} \times L \geq \overline{p}_{2l} \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_u - t_{ref}\right) - \overline{v}_{max} \times L \leq \overline{p}_{2u} \end{cases} \tag{2.2}$$

2.4.6.1 Time-slice Queries

For time-slice queries, $\overline{p}_{1l} = \overline{p}_{2l}, \overline{p}_{1u} = \overline{p}_{2u}$, and $t_l = t_u$, Eqn. (2.2) effectively becomes:

$$\begin{cases} \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L \geq \overline{p}_l \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L \leq \overline{p}_u \end{cases} \tag{2.3}$$

The query region for the one-dimensional time-slice query $Q1$ shown in Figure 2.1 is

illustrated in Figure 2.4. Points a and b are obtained by plugging $V = 0$ into Eqn. (2.3), and points c and d are obtained by plugging $V = 2v_{max}$ into Eqn. (2.3).

2.4.6.2 Window Queries

For window queries, $\overline{p}_{1l} = \overline{p}_{2l}$, $\overline{p}_{1u} = \overline{p}_{2u}$, Eqn. (2.2) effectively becomes:

$$\begin{cases} \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L \geq \overline{p}_l \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L \leq \overline{p}_u \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_u - t_{ref}\right) - \overline{v}_{max} \times L \geq \overline{p}_l \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_u - t_{ref}\right) - \overline{v}_{max} \times L \leq \overline{p}_u \end{cases} \tag{2.4}$$

The query region for the one-dimensional window query $Q2$ from Figure 2.1 is illustrated in Figure 2.5. The values for the points a, b, c, and d are obtained by plugging $V = 0$ into Eqn. (2.4), and the values for the points a', b', c', and d' are obtained by plugging $V = 2v_{max}$ into Eqn. (2.4).

2.4.6.3 Moving Queries

Figure 2.6 illustrates the query region for a one-dimensional moving query, using query $Q3$ from Figure 2.1 as an example.

In all cases, the query region for one-dimensional queries is a bounded polygon that is confined within an upper bound and a lower bound. Note that the upper bound and the lower bound are not necessarily straight lines (refer to Figures 2.5 and 2.6), since we take into consideration the case where objects move in opposite directions. We thus define the query region with six points, $U1$, $U2$, $U3$, $L1$, $L2$, and $L3$ in Figure 2.6, among which the four marginal points $U1$, $U3$, $L1$, and $L3$ are obtained by calculating intersections of the four query region boundary lines (which are produced by setting the comparison in Eqn. (2.4) to equals), with the boundaries of the underlying dual transformed space.

$L2$ is obtained by calculating the intersection of the following set of lines:

$$\begin{cases} \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L = \overline{p}_{1l} \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L = \overline{p}_{2l} \end{cases} \tag{2.5}$$

$U2$ is obtained by calculating the intersection of the following set of lines:

$$\begin{cases} \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L = \overline{p}_{1u} \\ \overline{P}_{ref} + \left(\overline{V} - \overline{v}_{max}\right)\left(t_l - t_{ref}\right) - \overline{v}_{max} \times L = \overline{p}_{2u} \end{cases} \tag{2.6}$$

In the case where either of $L2$ or $U2$ is outside the boundaries, the end points are used.

Effectively, a d-dimensional query body consists of d such distinctive query regions corresponding to the d dual transformed planes.

2.4.6.4 STRIPES Search Algorithm

Queries are processed in STRIPES as follows: At level l, each of the f grids are tested for relative position to the query body. This test is performed as a conjunction of d two-dimensional relative position tests between data regions and the corresponding query region. Relative positions include INSIDE, OVERLAP, and DISJUNCT. A grid is INSIDE a query body if and only if all the sub-queries return INSIDE; it is DISJUNCT as soon as one of the sub-queries returns DISJUNCT; otherwise OVERLAP is returned. For all the grids that return an INSIDE result, we immediately retrieve the entries within. DISJUNCT results are discarded and OVERLAP results are further probed recursively. Figure 2.7 shows the algorithm for relative position test between a data region and a query region. Figure 2.8 shows the relative positions between data regions and the query region. As shown in Figure 2.8, $R3$ is DISJUNCT to the query region, while $R2$ is INSIDE the query region and $R1$ OVERLAPs the query region.

An additional optimization technique that we use is based on the following observation. The $2d$-dimensional grid with each of its d 2-dimensional planes partitions the data space

```
Algorithm RelativePosition(R, U1, U2, U3, L1, L2, L3)
if (lowerLeftVertex(R) is on or above LineSegment(L1, L2) and
   lowerLeftVertex(R) is on or above LineSegment(L2, L3) and
   upperRightVertex(R) is on or below LineSegment(U1, U2) and
   upperRightVertex(R) is on or below LineSegment(U2, U3)
 then return INSIDE
else if (lowerLeftVertex(R) is above LineSegment(U1, U2) and
        lowerLeftVertex(R) is above LineSegment(U2, U3)) or
        (upperRightVertex(R) is below LineSegment(L1, L2) and
        upperRightVertex(R) is below LineSegment(L2, L3))
then  return DISJUNCT
else return OVERLAP
```

Figure 2.7: Algorithm to Test the Relative Positions of Data and Query Regions

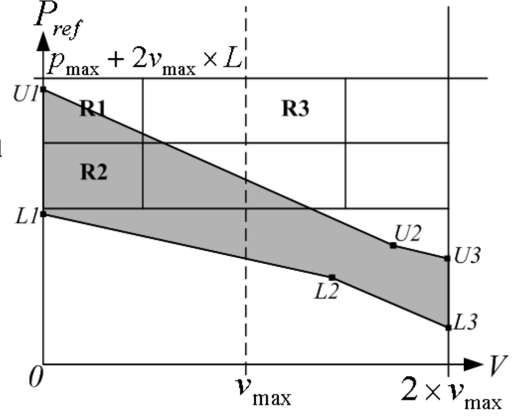


Figure 2.8: Relative Positions of Data and Query Regions for One-dimensional Points

into quads in each of the d planes. For any node, if any of these quads is DISJUNCT from the query region, then we can safely discard all nodes that are mapped to this disjunct quad. Effectively, whenever such quads are determined, number of search nodes that must be examined is reduced by 25%! This optimization technique quickly prunes away unnecessary node accesses, making the search very efficient.

2.5 Experimental Evaluation

In this section, we present results comparing the performance of the STRIPES and the TPR*-tree index.

2.5.1 Implementation Details and Experimental Platform

We implement both STRIPES and TPR*-tree [105] on top of the SHORE storage manager [22]. We compiled the storage manager with a 4KB page size. In all our experiments, we set the buffer pool size to 2048 pages; in making this choice for a small buffer pool size, we are essentially following the same philosophy as in previous studies [64, 72, 105] with the same goal of keeping the experiments manageable. SHORE pointers are 16 bytes in size, and we use 4-byte floating points for all the coordinate representation in the system (the TPR-tree code [72] also used floating point numbers).

The TPR*-tree is implemented using the algorithms described in [105]. The insert algorithm used in the TPR*-tree employs a priority queue that is implemented using heapsort (used in the *ChoosePath* algorithm in [105]). The priority queue stores the cost degradation for each node to insert the update. This queue is then used to determine the best node for inserting an update. The *PickWorst* algorithm of [105] is used to deal with overflow nodes. The best possible entries are removed and then reinserted again. Any possible overflow nodes are then split. The index is optimized for static point interval query which is same as the one used in TPR*-Tree paper.

With the system settings used in our experimental platform using SHORE, the maximum fanout of a TPR*-tree non-leaf node is 78.

For the STRIPES index, we simply create non-leaf nodes as (small) SHORE records. Since all sibling non-leaf nodes for a given parent are created concurrently, these nodes are usually stored sequentially on disk. This clustering property results in efficient disk access for the non-leaf nodes. To implement the leaf node, we use two leaf node sizes, which in the following discussion are referred to as small and large. When a leaf node is first created, its size is set to small, which is approximately half a disk page size. When a small leaf node overflows, it is promoted to a large node. Large nodes occupy exactly one disk page. We adopt this strategy since a split of a leaf node results in the creation of 16 new leaf nodes (for objects moving in two dimensions). In practice we have found that many of these leaf nodes are empty, and we don't create disk pages for these nodes during the split. Nevertheless the leaf page occupancy is still low at around 12%. Using the two leaf node size allows us to nearly double this page occupancy. With this implementation we find that the STRIPES index is about 2.4 times larger than the TPR*-tree index. In the future, we plan on extending our current implementation to use more than two leaf node sizes, which will increase the occupancy of the leaf-nodes further. However, based on current experimental evaluation, we expect that this may have limited additional benefit on the actual performance of the index as the index size is an issue only in very limited

cases. The key to the performance of STRIPES comes from having a relatively small disk footprint for the non-leaf nodes, which results in significant performance advantages over the TPR*-tree index. As an example, for a data set with 500K users, the TPR*-tree index has a height of four and the index occupies around 4,600 disk pages; whereas, the STRIPES index has a maximum height of seven and occupies around 11,200 pages. For this data set the STRIPES index has only 1, 486 non-leaf nodes. Each non-leaf node uses 352 bytes for its disk representation, which allows for around 11 non-leaf nodes to fit on a single disk page. Even as the index is updated over time, the non-leaf nodes are contained within a few hundred pages.

The experimental platform used in these experiments is a 2 GHz Intel Xeon machine with a 512KB L2 cache, a 40GB Western Digital 7200 RPM IDE Hard Drive, running Red Hat Linux 9.

2.5.2 Data Sets and Workload

We generated a number of workloads using the popular workload generator, which is generously provided by the inventors of the original TPR-tree [72]. This workload generator simulates objects moving in a two-dimensional space, and has a number of different parameters which can be varied. Although we experimented with a wide range of workloads with different combinations of parameter values, in the interest of space, in this section we only present results from using a few representative workloads. These workloads closely correspond to the default values used in the generator, which essentially generates the key data sets used in [72]. In the following paragraphs, we describe the key parameters of this workload generator, and also specify the values for these parameter that we used for generating our workloads.

The workload generator of Šaltenis et al. [72] allows generation of both uniform data workloads, and skewed workloads. In skewed workloads, two-dimensional objects move in a network of routes connecting a number of destinations, ND. As the value of ND

decreases, the skew in the data increases. In our experiments, we generate skewed data sets with ND = 20, 40 and 60.

In our workloads, we vary the number of moving objects, N, from $100K$ to $900K$. For the $100K$ data set, the objects move around in a space with dimensions of 1000×1000 kilometers. For larger data sets, we scale the dimensions to keep the densities same across all data sets (this strategy for generating scaled data sets is also recommended by [72]). For the uniform workloads, the initial positions of objects are uniformly distributed in space. The workload generator assigns initial positions for each moving object in the system, and then generates a workload which is a mix of update and query operations. The ratio of the number of update and query operations can be varied, and we present results using a mix of $80-20$, $50-50$, $20-80$. For the $80-20$ case, 80% of the operations are updates and 20% are queries.

For updates, the directions of the velocity vectors are assigned randomly. The default values for speeds are uniformly distributed between 0 and $3km/min$. The rate of updates is controlled by a parameter, called the update interval, UI. The time interval between successive updates is uniformly distributed between 0 and $2UI$. In the experiments presented in this section, we set UI to the default value of 60. The workloads are generated for the default 600 time units.

For the queries, the generator can generate any arbitrary mix of time-slice, window, and moving queries. The default values for the query mix are 60%, 20% and 20%; all workloads used in this chapter are generated using this default setting. The temporal range of the queries is set to the default value of 40, and the spatial part of the queries is set to the default value of 0.25% of the entire spatial extent.

2.5.3 Effect of Workload Mix

In this first experiment, we use a uniform data set with $500K$ moving objects. The first experimental result for this data set is shown in Figure 2.9. In this figure, we plot the total

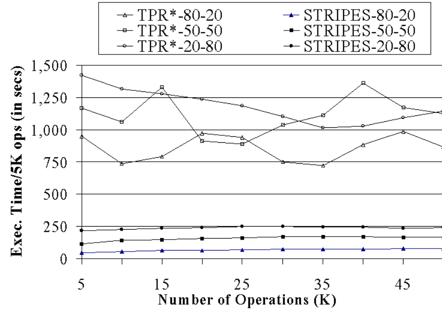


Figure 2.9: 500K-Uniform: Continuous Performance Measurement for Batch of 5K Operations

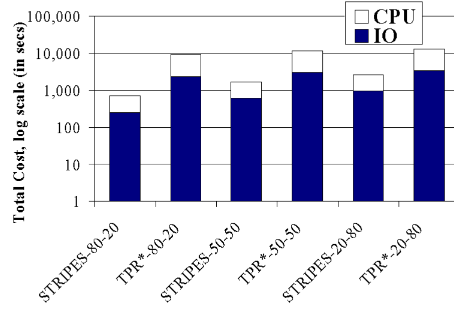


Figure 2.10: 500K-Uniform: I/O and CPU Costs for 50K Operations

execution time for the two index structures shown in batches of $5K$ operations for the first $50K$ operations. This experiment lets us determine if the performance of the indices deteriorates as the updates operations change the underlying partition boundaries in the indexing structures.

As can be seen from Figure 2.9, the TPR*-tree index has a fairly good steady state behavior. This result is consistent with the results presented in the [72]. (In [72] the researches also show that in contrast to the TPR*-tree, the performance of the original TPR-tree rapidly degrades for a similar experiment.) The TPR*-tree has a good steady state behavior since it uses a much more sophisticated update algorithm (the *ChoosePath* component), which prevents the R*-tree from getting into situations when increasing amounts of dead space and overlap amongst the bounding boxes lead to a rapid drop in performance.

From Figure 2.9 we observe that STRIPES also demonstrates good steady state behavior. Furthermore, STRIPES is at least **4x faster** than the TPR*-tree index! The reason for this efficiency is that the non-leaf nodes of the STRIPES index occupies only a few hundred pages even as the indexing structure changes with new updates. These nodes are typically resident in the buffer pool and I/Os are usually only needed for accessing the leaf-pages. In contrast, during an insert operation in the TPR*-tree, multiple paths are traversed down the tree in the *ChoosePath* algorithm, which results in a large number of I/Os.

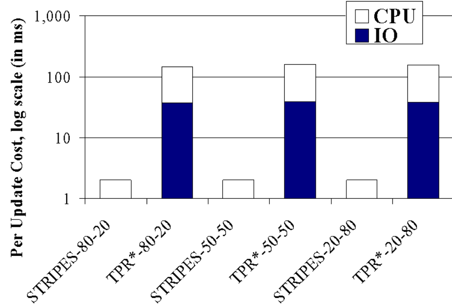


Figure 2.11: 500K-Uniform: Average Single Update Costs

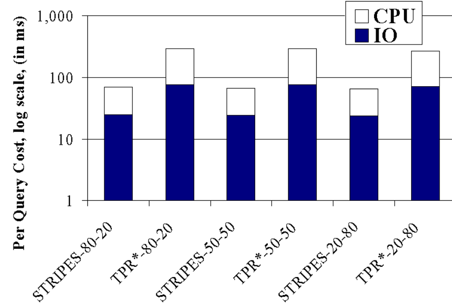


Figure 2.12: 500K-Uniform: Average Single Query Costs

We see these results more clearly in Figure 2.10, which breaks down the total costs for the first 50*K* operations into the CPU and the I/O components. To produce this cost breakup, we tracked the time spent in I/O operations, and used this measure to divide the total execution time into the I/O and the CPU components[1] Note that in this figure the *y*-axis uses a log-scale. As shown in Figure 2.10, the I/O costs for TPR*-tree are very high relative to the I/O cost for STRIPES. This is because the STRIPES index only requires a handful of I/Os for every update operation. For this data set (with 500*K* objects), the STRIPES non-leaf nodes are usually spread across a few hundred disk pages. These are usually resident in the buffer pool, and I/Os are only needed for the leaf-level pages. In contrast, the TPR*-tree index incurs a large number of I/Os. In this case, the index occupies around 4,600 disk pages and the index is of height 4. During the insert operation, the *ChoosePath* algorithm has to find a good leaf node for the insertion. To accomplish this task, it uses a priority queue based technique to traverses multiple paths to the leaf nodes (see Section 2.3.2). This technique results in large number of I/Os, and also leads to poor reference locality as successive updates are likely to traverse different parts of the index!

For this workload, we also plot the average cost for a single update operation in Figure 2.11, and the average cost for a single query in Figure 2.12. As can be seen from

[1]To accomplish this task, we turned off asynchronous I/Os that are incurred by the SHORE background cleaner thread, and used only one thread to carry out all the workload operations.

these figures, the I/O cost for the TPR*-tree index is significantly higher than the I/O cost incurred by STRIPES for *both* the update and query operations. The difference is much more dramatic for the update operation, which is extremely efficient in STRIPES (see Figure 2.11). Update operations in both index structures require an insert operation. An insertion in STRIPES only requires inserting a point object, which can be accomplished by a *single* path traversal from the root (see Section 2.4.3). This operation is extremely fast in quadtree based structures because of the non-overlapping regular decomposition strategy used by the index structure. In contrast, multiple paths are traversed by the TPR*-tree, which results in a much higher I/O cost.

Figure 2.11 and Figure 2.12 also show that the CPU costs incurred by STRIPES is much lower than the CPU costs for the TPR*-tree index. For updates, the reason for this is once again the efficiency of the update operation in STRIPES, as compared to the much more expensive technique of multiple path traversals used by the TPR*-tree, which require expensive overlap comparisons at each node. In addition, the CPU costs for the TPR*-tree insert also includes the sort cost and reinsert algorithm ($pickWorst$) cost. Calculation of the integrals needed for the TPR*-tree are also expensive and contribute to the high CPU cost.

For queries, the techniques employed by STRIPES (descri-bed in Section 2.4.6.4) are much more CPU efficient as compared to the overlap comparisons that are need in the TPR*-tree.

2.5.4 Scaling with Increasing Number of Moving Objects

In this experiment we explore the effect of increasing the number of moving objects from $100K$ to $900K$ users for the three workloads ($80-20, 50-50$, and $20-80$). In the interest of space we only present results for the $50-50$ case for $100K$ and $900K$ data set cardinalities. These results are shown in Figure 2.13 as per query and update costs, broken down by the CPU and I/O costs.

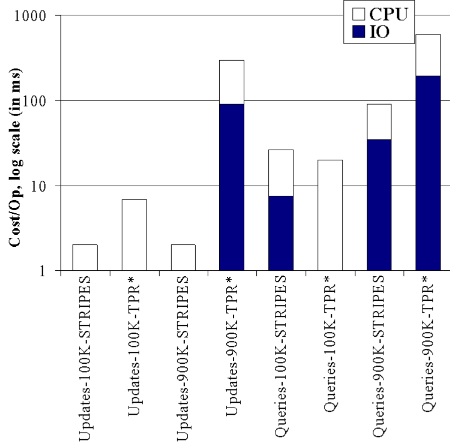


Figure 2.13: Effect of Number of Moving Objects

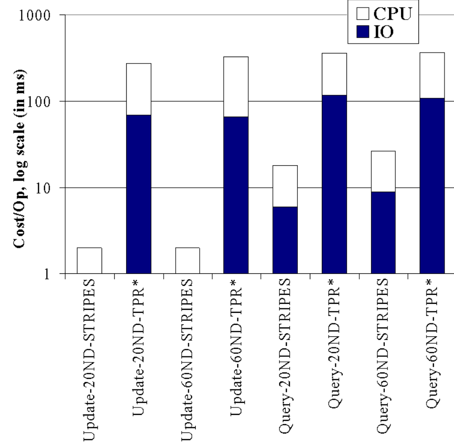


Figure 2.14: 500K-Skewed: Single Query Costs

For the 100*K* data set the TPR*-tree index fits in the buffer pool and incurs no I/O cost, whereas STRIPES incurs I/Os, especially for the queries. For this case the smaller index size of TPR*-tree works to its advantage, and the query performance of STRIPES is about 35% worse than the query performance of the TPR*-tree index. Aggressive disk space optimization outlined in section 2.5.1, are may improve the performance of STRIPES in this case, and we plan on undertaking this effort as part of our future work.

Note that even in the case with the 100*K* data set, the update operation in STRIPES is about **5x faster** as compared to the TPR*-tree. Again the reason for this performance gap is the difference between the expensive insert operation in TPR*-tree and the highly efficient insert operation in STRIPES.

For the 900*K* data set the performance gap between the two indices widens even further from what we observed with the 500*K* data set in Section 2.5.4. The reasons for this follow from the discussion in Section 2.5.4 as the STRIPES index keeps most of its non-leaf nodes resident in memory even for this larger data set. The TPR*-tree's insert algorithm degrades even further because of the larger data set.

2.5.5 Effect of Data Skew

In this final experiment, we evaluate the performance of the two indices for skewed data sets. We experimented with the ND parameter in the workload generator (see Section 2.5.3) and generated data sets with $ND = 20$, 40, and 60. In the interest of space we only present the results for $ND = 20$ (highly skewed) and $ND = 40$ (small skew) data set, for only the $50 - 50$ workload. Figure 2.14 plots the per update and per query costs for this experiment. Comparing the update and query costs in this figure with the costs for the $50 - 50$ workload in Figure 2.11 and Figure 2.12, we can observe that both index structures handle skewed data sets well, and *STRIPES continues to outperform the TPR*-tree by over an order of magnitude for updates, and by* ***4x*** *for queries.*

2.5.6 Summary

In summary, we have shown through extensive experimental evaluation that STRIPES is significantly faster than the TPR*-tree index. The update operation in STRIPES is often more than an order of magnitude faster, and the query performance is around 4x faster as compared to the TPR*-tree. The regular disjoint decomposition of space that is used by STRIPES results in extremely efficient inserts. In addition, even with very large data sets (relative to the available buffer pool size), the amount of space needed to hold the non-leaf nodes of STRIPES is very small. Consequently, I/Os are rarely incurred for the non-leaf nodes. In contrast the TPR*-tree suffers from having to traverse multiple paths down the index, which is I/O intensive and results in a reference pattern that has poor cache locality.

These differences in the indexing approaches also manifest in the CPU costs as the TPR*-tree has to carry out many expensive box overlap computations as it traverses down the index. In contrast STRIPES employs a number of optimizations (refer to Section 2.4.6.4) to keep CPU costs low.

2.6 Related Work

Within the broader context of indexing trajectories for moving objects, there are two broad classes of related works. The first class includes methods for indexing the historical and the current positions. The second, and more closely connected class of research to our work, is on indexing the predicted locations of moving objects. The methods for indexing the past and the current locations are typically concerned with queries on exact trajectory points, whereas methods for indexing on the future locations are concerned primarily with indexing the parameters of the predicted trajectory representations, which typically include a velocity vector and a start position vector. However, both these classes of indices are concerned with efficient indexing mechanisms for supporting fast updates and queries on spatial representations of the trajectories. In the next paragraph we briefly review the methods for indexing on the past trajectory locations, and then turn our attention to the more closely related work in indexing predicted trajectories.

Most of the work on indexing the past locations of trajectories is based on variations of the R-tree [44] and the R*-tree [10]. These methods include the 3-D R-trees [108] which simply treats time as a third dimension. The MR-tree [113] and the HR-tree [76] are also 3-D R-tree structures and maintain a separate R-tree for each time stamp. The MV3R-tree [104] is a hybrid structure that uses a multi-version R-tree (MVR) for time-stamp queries, and a small 3D R-tree for time-interval queries. This indexing structure has been shown to outperform other historical trajectory indexing structures, such as the popular TB-tree [85]. SEB [98] and SETI [23] are historical trajectory indexing techniques that partition the spatial extents, and build indices on the temporal dimension. A number of indexing methods have also focused on efficient methods for indexing the current location of moving objects [61, 62, 75, 97]. All the methods described in this paragraph are not concerned with indexing the predicted location, and index the native space of the trajectories. In contrast, STRIPES indexes the predicted locations in dual transformed space.

Two main approaches have been used for indexing the *predicted* locations of trajectories. These two approaches are a) methods that index the predicted trajectories in the original spatial and temporal dimensions, and b) methods that transform the predicted trajectories into a dual transform space and index the dual transformed space.

One of the early works on indexing predicted trajectories is by Tayeb et al. [106]. In this work, trajectories in a d-dimensional space are treated as lines in a $(d + 1)$-dimen-sional space, with time as the additional dimension. The line is then indexed using a PMR quadtree [93]. The drawbacks of this approach are that the index may have excessive dead space and replication since it is indexing high dimensional lines. The work by Tayeb et al., carried out within the context of the MOST [93] project, has strongly influenced and stimulated interests in methods for querying moving object databases.

The TPR-tree [72] is a popular indexing structure for indexing predicted trajectories. This index structure uses the basic R-tree indexing structure and extends the notion of bounding boxes to time-parameterized bounding boxes as described in Section 2.3.1. The notion of time-parameterized bounding box has also been used by other related indexing structures [21, 87]. One of the problems with the time-parameterized boxes is that estimating it requires reasoning about the positions of the objects enclosed by the box over some period of time. The original TPR-tree paper [72] used a conservative bounding box, but this has been improved in a number of different ways [70, 88, 90], often by exploiting various additional parameters such as expiration times or the maximum speed. The TPR*-tree is an index structure which improved the methods proposed in the original TPR-tree, and has been shown to be significantly faster than the TPR-tree. In this paper we compare STRIPES with the TPR*-tree, and show that STRIPES outperforms the TPR*-tree by very significant margins. An extensive critique of the TPR*-tree can be found in [42]. Dual transformation techniques have been successfully employed for querying static spatial data [51]. Drawing inspiration from this success, dual transformation techniques have also been proposed for indexing predicted trajectories [111]. These indexing methods include

the Kinetic data structure [6], the R-tree based parameterized space indexing method [87], and the SV-model [25]. Perhaps the most popular dual transformation approach for predicted trajectories is the work by Kollios et al [59]. In this work, the authors derive nice lower bounds on the cost of answering predictive queries using dual transformation. Most of the paper is concerned with objects moving in one-dimensional space, and the paper sketches extensions to higher-dimensional space. In addition, the paper only considers window queries. The largely theoretical approach has served as the basis for some of the choices made in the TPR-tree [72], but has largely been dismissed by recent work that use a more systems-approach [72, 105]. The dual transformation method used in STRIPES is based on the Hough-X transform used in [59]. STRIPES can handle the entire range of predictive queries, including moving window queries, and we show that STRIPES vastly outperforms the current best know method for indexing predicted trajectories. Immediately following the STRIPES work, a B^+-tree based dual transform indexing technique called the B^x-tree [54] was also proposed for indexing predicted trajectories. However, no performance comparison study has been done to evaluate the efficiency of STRIPES against the B^x-tree yet.

In recent years, a few motion modeling approaches have also been proposed for indexing historical (the PA-tree [77]) and predicted (the STP-tree [103]) trajectories. Both the PA-tree and the STP-tree use complex polynomial approximations to model trajectories and use existing index structures to index coefficients derived from the polynomial approximations.

For a more detailed overview of related work in this area prior to STRIPES, the reader is directed to a comprehensive review [74].

2.7 Conclusions

In this chapter we have presented a new indexing structure called STRIPES for indexing and answering queries on predicted positions in moving object databases. This new

indexing structure draws inspiration from earlier largely theoretical work in this area, advocating the use of dual transformation for indexing such data sets. The STRIPES index leverages these dual transformation techniques and uses a disjoint regular partitioning technique to efficiently index the points in the dual transformed space. The STRIPES index can support all the types of commonly used predictive queries [72], which include time-slice, window, and moving queries. We have compared the performance of STRIPES with the most efficient predictive indexing structure, the TPR*-tree [105]. Our comprehensive experimental evaluations demonstrate that STRIPES outperforms the TPR*-tree index for both updates and queries; updates are often more than an order of magnitude faster using STRIPES, and queries are often faster by a factor of 4x. These differences can be seen in both the I/O and the CPU costs. Consequently, STRIPES is an extremely efficient and practical indexing structure for supporting predictive queries.

CHAPTER 3

ALL NEAREST NEIGHBOR QUERY ALGORITHMS AND METRICS

3.1 Introduction

The All Nearest Neighbor (ANN) operation takes as input two sets of multi-dimensional data points and computes for each point in the first set the nearest neighbor in the second set. The ANN operation has a number of applications in analyzing large multi-dimensional datasets. For example, clustering is commonly used to analyze large multi-dimensional datasets, and algorithms such as the popular single-linkage clustering method [52, 56] uses ANN as its first step. A related problem, called AkNN, which reports the kNN for each data point, is directly used in the Jarvis-Patrick Clustering algorithm [53]. AkNN is also used in a number of other clustering algorithms including the k-means clustering and the k-medoid clustering algorithms [17].

The list of applications of ANN and AkNN is quite extensive and also includes co-location pattern mining [114], graph based computational learning [58], pattern recognition and classification [78], N-body simulations in astrophysical studies [31], and particle physics [79].

ANN is a computationally expensive operation ($O(n^2)$ in the worst case), and its cost increases rapidly with increasing dataset sizes. In many applications that use ANN, especially large scientific applications, the datasets are growing rapidly and often the ANN computation is one of the main computational bottlenecks. Recognizing this problem, there has been a lot of interest in the database community in developing efficient external

ANN algorithms [17, 18, 29, 46, 116]. All of these methods build R*-tree indices [10] on one or both datasets, and evaluate the ANN by traversing the index. During the index traversal, these methods keep track of nodes in the index that need to be considered, and employ a priority queue (PQ) to determine the order of the index traversal. The efficiency of these algorithms strongly depends on how many PQ entries are created and processed. The most common and effective pruning method that has been developed so far employs a pruning metric called MAXMAXDIST, which is roughly the maximum distance between any points in two minimum bounding rectangles (MBR). This pruning metric can be used to guarantee that certain subtrees in the index will not produce a nearest-neighbor (NN), and hence can be pruned out from further consideration. In this chapter we introduce a new distance metric, called the MINMAXMINDIST (abbreviated as NXNDIST), and show that this new metric has a much more powerful pruning effect. Using extensive experiments we show that *this new distance metric often improves the performance of ANN operation by more than an order of magnitude*.

In this chapter we also explore the properties of NXNDIST and develop a fast algorithm for computing this metric. This fast algorithm is critical since for ANN queries this distance computation is evaluated frequently.

In addition, we examine a family of index based ANN algorithms, which differ in the way that the spatial indices are traversed, and the way in which the PQ entries are expanded. We explore four options corresponding to two forms of tree traversal – breadth-first and depth-first, and two forms of PQ entry expansion – expand both index nodes in the priority queue entry or expand only one node at a time. To the best of our knowledge, no previous work has systematically explored these alternatives in the context of ANN evaluation. A contribution of this chapter is the exploration of these alternatives. More importantly, we show that of the four algorithms in this design space, *the depth-first bi-directional expansion method is consistently the most efficient*.

All of the previous index based ANN methods [17, 18, 29, 46, 116], have used the

"ubiquitous" R*-tree index as the indexing structure. In this chapter, we show that for ANN queries there is a much better choice for an index structure. This new indexing structure is called the MBRQT index, and is essentially a disk based bucket PR quadtree [93], with the addition of the MBR (minimum bounding rectangle) information for each internal node. We show that the regular decomposition and non-overlapping nature of the quadtree results in a much more effective pruning strategy for ANN computation. Our experimental results show that *ANN evaluation using MBRQT is around 3X faster than using R*-tree.*

Besides comparing our methods with previous index based ANN methods, we also extensively compare our methods with the GORDER [112] ANN method. Unlike other methods, GORDER doesn't employ an index to speed up the ANN computation. Instead it first transforms the data using Principal Components Analysis and imposes a grid structure on the transformed space. Then, it "joins" the two datasets using the grid structure, carefully exploring only a limited number of grid cell pairs. The GORDER [112] approach has not been compared with BNN [116], which is currently the best index based ANN method, and in this chapter we compare our technique with both these previous methods. These comparisons show that our method significantly outperforms both these previous methods.

We note that quadtree structures are not height-balanced, but methods using a disk based structure for the quadtree have been shown to be effective [33, 47] for spatial database applications. The method using MBRQT can be used in cases where the database system chooses to support quadtrees (for example, Oracle has support for traditional quad-trees [60]), or in cases where ANN is run on datasets that do not have a prebuilt index (such as when running ANN as part of a complex query in which a selection predicate may have been applied on the base datasets).

The remainder of this chapter is organized as follows: Section 3.2 covers related work. Section 3.3 outlines our new ANN approach. Section 3.4 contains a comprehensive experimental evaluation of our new approach, and compares it with previous methods.

Finally, Section 3.5 contains our conclusions.

3.2 Related Work

The problem of Nearest-Neighbor (NN) to a query point has been well studied from a database query processing perspective [46, 82, 92]. These methods use an R*-tree index for evaluating NN. Essentially these methods develop various strategies for traversing the index by using a priority queue (PQ) to record and order the index nodes that must be traversed. Usually, methods for pruning the PQ entries are also used to discard portions of the index that are guaranteed not to contain the NN. The earliest of these methods by Roussopoulos et al. [92] introduces two key metrics between a point and an MBR, called MINDIST and MINMAXDIST, which are used in producing an efficient traversal. The MINDIST is the minimum distance between the query point and an MBR entry of the index, and the MINMAXDIST is the minimum value of the maximum distance between the query point and the points on the edges of an MBR. Essentially MINDIST is an optimistic NN bound and MINMAXDIST is a pessimistic one. (These two key metrics have also been often used in problems related to NN.) The method in [92] uses a depth-first traversal, which was later shown to be suboptimal [82]. An I/O optimal algorithm for NN search was later provided [46], which essentially employs a breadth-first (BF) search technique to traverse the R*-tree index. Böhm et al. [15] give a comprehensive comparison and coverage of different structures and techniques that address NN query processing.

Distance join algorithms are also related to ANN algorithms [46]. A distance join operation works on two sets of spatial data, and computes all object pairs, one from each set, such that the distance between the two objects is less than a non-negative value d. A distance semi-join is a related operation [46], which essentially produces one result per entry of the outer relation. Incremental algorithms for these operations are also developed [46]. Later, Shin et al. [95] develop a more efficient algorithm for a related problem of k-distance join, which uses a bi-directional expansion of entries in the PQ and

a plane-sweep method.

With respect to how results are produced, Corral et al. [28, 29] propose both iterative and recursive non-incremental distance join methods. These methods employ more efficient pruning techniques and are thus more effective, while incremental algorithms are more flexible and better suited for online query processing.

The closest body of related work is the collection of previously proposed external memory ANN algorithms. A simple approach for computing ANN is to run a NN algorithm on the inner dataset **S** for each object in the outer dataset **R**. For this approach, optimization techniques have also been proposed to reduce CPU and I/O costs [19]. However, the assumption for such optimization is that the queries fit in main memory, which makes it inefficient when the size of **R** is larger than the main memory size.

Depending on whether **R** and/or **S** are indexed or not, existing techniques fall into two categories: traversal of R*-tree indices using a Distance Join algorithm [29, 46], and hash based algorithms using spatial partitions [38]. The work in [116] spans both categories. Böhm and Krebs [18] also provide a solution to the more general problem of *Nearest Neighbor Join*: namely find for each object in **R**, its k nearest neighbors in **S**, which degenerates to ANN when $k = 1$. However, a specialized index structure termed *multipage index* is proposed for the solution provided, and thus the solution in [18] does not apply to general-purpose index structures such as R*-trees or quadtrees.

Incremental Distance Join algorithms have also been used to evaluate ANN queries [29, 46]. However, in the case where some of the objects in **R** have nearest neighbors with large distances, these algorithms incur significant overhead, as more entries than necessary will have to be processed before the NNs for those objects are identified.

The more recent work on ANN by Zhang et al. [116] suggests two approaches to the ANN problem when the dataset **S** is indexed: *Multiple nearest neighbor search (MNN)*, and *Batched nearest neighbor search (BNN)*. MNN is essentially an index-nested-loops join operation, where the locality of objects is maximized to minimize I/O. However, the

CPU cost is still high because of the large number of distance calculations for each NN search. To reduce the CPU cost, BNN splits the points in **R** into n mutually exclusive but overall exhaustive groups, and traverses index **S** only n times, greatly reducing the number of distance calculations.

For the case where neither dataset has an index, Zhang et al. [116] also propose a hash based method (HNN) using spatial hashing introduced in [84]. However, it was pointed out that in many cases building an index and running BNN is faster than HNN, and HNN is also susceptible to poor performance on skewed data distributions [116].

The recent GORDER [112] method also takes an approach similar to [116]. However, GORDER employs a Principal Components Analysis (PCA) technique to transform the union space of the two input datasets to a single principal component space, and then sorts the transformed points using a superimposed *Grid Order*. The transformed datasets, often more uniformly distributed, are written back to disk in sorted order. A Block Nested Loops join algorithm is then used for solving the KNN join query.

The BNN and the GORDER approaches are currently regarded as highly efficient ANN methods. To the best knowledge of the authors, previous work has not compared these two methods directly. In this chapter we make this comparison, and also compare these two methods with our new techniques.

Interestingly, previous research on ANN and related join methods (such as Distance Join) has not considered the use of disk-resident quadtree indices. As we show in this chapter, the regular decomposition and non-overlapping properties of the quadtree make it a much more efficient indexing structure for ANN queries.

3.3 ANN Evaluation

In this section, we first introduce a new asymmetric distance metric, MINMAXMINDIST (abbreviated as NXNDIST), which has a higher pruning power for ANN computation compared to the traditional MAXMAXDIST metric. We also present an efficient

Table 3.1: Table of Frequently Used Notations

Notation	Description
D	Dimensionality of data space
R	Query object dataset
S	Target object dataset
I_R	Index on dataset **R**
I_S	Index on dataset **S**
M	An MBR in index I_R
N	An MBR in index I_S
r	Point object in the dataset **R**
s	Point object in the dataset **S**

algorithm for computing NXNDIST that has linear cost with respect to dimensionality. Next, we explore the family of four ANN algorithms, namely: breadth-first search with bi-directional node expansions (ANN-BFBI); breadth-first search with uni-directional node expansions (ANN-BFUNI); depth-first search with bi-directional node expansions (ANN-DFBI); depth-first search with uni-directional node expansions (ANN-DFUNI), together with the pruning heuristics that take advantage of some of the inherent properties of the NXNDIST metric for more effective pruning.

We then present a generalization of our method for handling AkNN search problems.

Finally, we propose a new index structure, which is called the Minimum Bounding Rectangle enhanced Quad-Tree (MBRQT), which has significant advantages over an R*-tree for ANN computation as it maximizes data locality and avoids the overlapping MBR issue that is inherent in an R*-tree index.

To facilitate our discussion, we will use the notations introduced in Table 3.1.

3.3.1 A New Pruning Distance Metric

As is common with current ANN algorithms, a certain distance metric is required as the upper bound for pruning entries from I_S that do not need to be explored. Traditionally the MAXMAXDIST metric has been used as such an upper bound [28, 29]. The

MAXMAXDIST between two MBRs is defined as the maximum possible distance between any two points each falling within its own MBR [28, 29]. We observe that the MAXMAXDIST metric is an overly conservative upper bound for ANN searches. We show that, for ANN queries a much tighter upper bound of the distance between points within M and those within N can be derived. This new upper bound guarantees the enclosure of the nearest neighbor within N for every point within M. We call this new metric the NXNDIST, and formally define it in the next section.

3.3.1.1 Definition and Properties of NXNDIST

For completeness and ease of comparison, first we provide brief descriptions of two related distance metrics on MBRs that have been previously defined [28]. These metrics are MINMINDIST and MINMAXDIST.

The MINMINDIST between two MBRs is the minimum possible distance between any point in the first MBR and any point in the second MBR. This metric has been extensively used in previously proposed ANN methods as the lower bound metric for pruning ANN processing. We also employ this metric as a lower bound measure (NXNDIST, which we define in this section, is our upper bound metric).

Another distance metric termed MINMAXDIST [28], is the upper bound of the distance between at least one pair of points, one from each of the two MBRs. MINMAXDIST has been frequently used as an upper bound pruning metric in various distance join algorithms (for example, [28, 29]). However, we note that MINMAXDIST was proposed to address a different class of distance join operations, and is not suitable as a pruning metric for ANN computation as it does not provide a correct upper bound for ANN.

In the following discussion, we define the NXNDIST metric in arbitrary dimensions and explore its properties.

We represent a D-dimensional MBR with two vectors: a lower bound vector to record

the lower bound in each of the D dimensions, and an upper bound vector to record the upper bound in each of the D dimensions. For example, the lower bound vector for an MBR M is expressed as $< l_1^M, l_2^M, ..., l_D^M >$, and the upper bound vector for M is represented as $< u_1^M, u_2^M, ..., u_D^M >$.

On the other hand, a D-dimensional point p is represented as the vector $< p_1, p_2, ..., p_D >$.

Next, we define a few auxiliary metrics, and then give the definition of the NXNDIST metric.

***Definition* 3.1.** *Given two D-dimensional points, p and q, $DIST_d(p, q)$ in dimension d is defined as:*

$$DIST_d(p, q) = |p_d - q_d|$$

***Definition* 3.2.** *Given two D-dimensional points, p and q, $DIST(p, q)$ is defined as:*

$$DIST(p, q) = \sqrt{\sum_{d=1}^{D} DIST_d^2(p, q)}$$

Definition 3.2 essentially gives the definition of the Euclidean distance between p and q.

***Definition* 3.3.** *Given two D-dimensional MBRs, M and N, for all points p enclosed in M and all points q enclosed in N, $MAXDIST_d(M, N)$ in dimension d is defined as:*

$$MAXDIST_d(M, N) = \max_{\forall p \in M, \forall q \in N} DIST_d(p, q)$$

In other words, $MAXDIST_d(M, N)$ gives the maximum distance between any points within M and those within N in dimension d. The geometric meaning of $MAXDIST_d(M, N)$ is as follows: in dimension d, starting at any point within M, an interval of extent $MAXDIST_d(M, N)$ in either direction is guaranteed to cover all points within N along this dimension.

***Definition* 3.4.** *Given two D-dimensional MBRs, M and N, and an arbitrary point p enclosed in M, $MAXMIN_d(M, N)$ in dimension d is defined as:*

$$MAXMIN_d(M, N) = \max_{\forall p \in M} (\min(|p_d - l_d^N|, |p_d - u_d^N|))$$

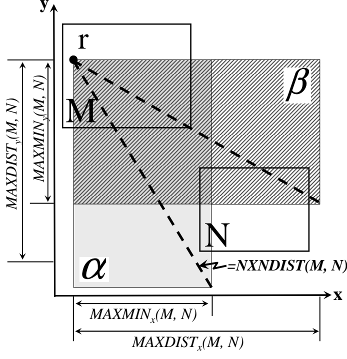


Figure 3.1: 2-D Intuition of NXNDIST

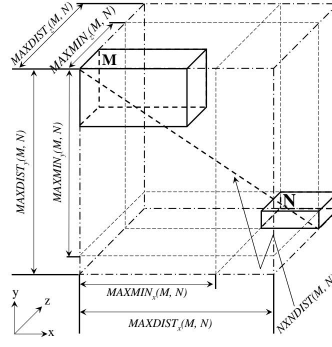


Figure 3.2: 3-D Computation of NXNDIST

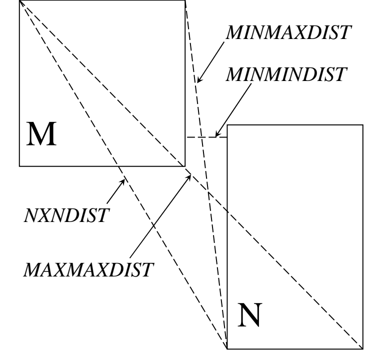


Figure 3.3: Metrics on MBRs

We give the intuitive definition of $MAXMIN_d(M,N)$ as "the maximum of the minimum distances in dimension d from any point within the range $[l_d^M, u_d^M]$ to at least one end point l_d^N or u_d^N".

***Definition* 3.5.** *Given two D-dimensional MBRs, M and N, $NXNDIST(M,N)$ is defined as:*

$$NXNDIST(M,N) = \sqrt{S - \max_{d=1,\ldots,D}\begin{pmatrix} MAXDIST_d^2(M,N) \\ -MAXMIN_d^2(M,N) \end{pmatrix}}, \textit{ where}$$

$$S = \sum_{d=1}^{D} MAXDIST_d^2(M,N)\,.$$

Figure 3.1 gives a geometric intuition of $NXNDIST(M,N)$ in 2-D space. Two non-overlaping MBRs M and N are shown, as well as an arbitrary point object $r \in M$. If an interval is constructed originating from r, with extent along the y axis equivalent to $MAXDIST_y(M,N)$ in either direction, then it is guaranteed to enclose N completely along the y axis. Sweeping the interval along the x axis with extent $MAXMIN_x(M,N)$, a rectangular search region is formed, which is the shaded region labeled α in the figure. As is shown in the figure, this rectangular search region is guaranteed to enclose at least one edge of N. Sweeping along the y axis in a similar fashion, a second search region β, which is shown as the hatched rectangle in the figure, can be also formed. Of the two search regions α and β, the shorter diagonal length is equivalent to $NXNDIST(M,N)$.

To generalize to D dimensions, the sweeping interval is replaced by a (D-1)

dimensional hyperplane, and there are a total of D different ways in which the sweeping can be performed. $NXNDIST(M, N)$ is then the minimum diagonal length among the D search regions.

A 3-D example of NXNDIST is depicted in Figure 3.2.

Figure 3.3 gives an illustration of two MBRs and various distance metrics between them.

Next, we prove the correctness of NXNDIST as an upper bound for ANN search as well as derive several lemmas that reveal some important properties of the NXNDIST metric.

***Lemma* 3.1.** *Given two MBRs, M and N, and a point object $r \in M$. Let $NN(r, N)$ denote r's nearest neighbor within N, then $DIST(r, NN(r, N)) \le NXNDIST(M, N)$.*

Proof. From the definition of $NXNDIST(M, N)$ (Definition 3.5), let i be the dimension in which

$$\begin{aligned} & MAXDIST_i^2(M, N) - MAXMIN_i^2(M, N) \\ = \; & \max_{d=1,\ldots,D}\left(MAXDIST_d^2(M, N) - MAXMIN_d^2(M, N)\right) \end{aligned}$$

Let p be a point enclosed in M. From the definition of $MAXMIN_i(M, N)$ (Definition 3.4), let q_i^N be the end point coordinate value of N in the ith dimension such that $\max_{\forall p \in M} |p_i - q_i^N| = \max_{\forall p \in M} (\min(|p_i - l_i^N|, |p_i - u_i^N|))$. For N to be a minimum bounding rectangle, there must exist within N such a point object s that $s_i = q_i^N$. Then from the definition of nearest neighbor, the following inequality holds:

$$DIST(r, NN(r, N)) \le DIST(r, s) \tag{3.1}$$

We observe the following inequalities from Definitions 3.3 and 3.4

$$DIST_i(r, s) \le MAXMIN_i(M, N) \tag{3.2}$$

$$\forall_{d=1}^{D} DIST_d(r, s) \le MAXDIST_d(M, N) \tag{3.3}$$

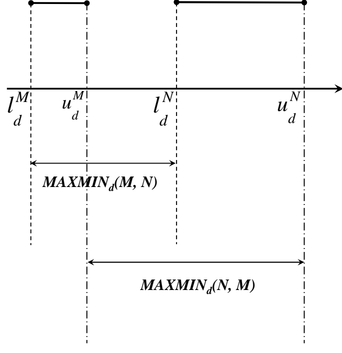


Figure 3.4: Counter-Example for MAXMIN

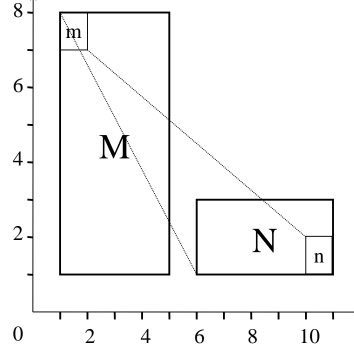


Figure 3.5: Counter-Example for NXNDIST

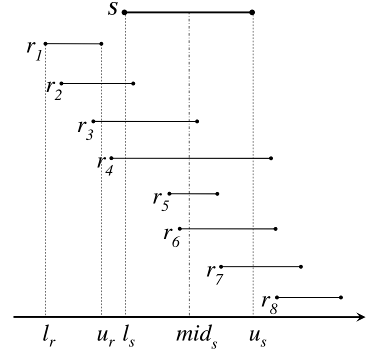


Figure 3.6: Computation of 1-D MAXMIN

We can also re-write the definition of $NXNDIST(M, N)$ as

$$NXNDIST(M, N) = \sqrt{\begin{array}{l} \sum_{d=1,\ldots,D}^{d \neq i} MAXDIST_d^2(M, N) \\ +MAXMIN_i^2(M, N) \end{array}} \tag{3.4}$$

It then follows from inequalities 3.2, 3.3 and equation 3.4 that

$$DIST(r, s) \leq NXNDIST(M, N) \tag{3.5}$$

From inequalities 3.1 and 3.5 we obtain $DIST(r, NN(r, N)) \leq NXNDIST(M, N)$. □

Lemma 3.1 establishes the foundation for the pruning heuristics presented in Sections 3.3.2.7 and 3.3.3.

***Lemma* 3.2.** *The MAXMIN metric is not commutable, i.e., given two D-dimensional MBRs M and N:* $MAXMIN_d(M, N) \neq MAXMIN_d(N, M)$.

Proof. Suppose that MAXMIN is commutable, that is:

$MAXMIN_d(M, N) \neq MAXMIN_d(N, M)$.

We provide a counter-example in Figure 3.4 as a trivial proof. It can be observed in Figure 3.4(a) that $MAXMIN_d(M, N) = l_d^N - l_d^M$, and $MAXMIN_d(N, M) = u_d^N - u_d^M$.

It is then straightforward to see that $MAXMIN_d(M, N) \neq MAXMIN_d(N, M)$. □

***Lemma* 3.3.** *The NXNDIST metric is not commutable, i.e., given two D-dimensional MBRs M and N:* $NXNDIST(M, N) \neq NXNDIST(M, N)$.

Proof. We observe that the derivation of $NXNDIST(M, N)$ is dependent on $MAXMIN_i(M, N)$ for some $i \in [0, D]$, and the derivation of $NXNDIST(N, M)$ relies on $MAXMIN_j(N, M)$ for some $j \in [0, D]$. Following the conclusion in Lemma 3.2, it is straightforward to see that NXNDIST is not commutable. □

***Lemma* 3.4.** *Let m be a child MBR of M, i.e.,* $m \subseteq M$ *then* $NXNDIST(m, N) \leq NXNDIST(M, N)$.

Proof. Consider the following informal proof by contradiction:

Suppose $NXNDIST(m, N) > NXNDIST(M, N)$. Then it follows that there exists some point $r \in m$ for which the following inequality holds:

$$DIST(r, NN(r, N)) > NXNDIST(M, N) \tag{3.6}$$

Since $r \in M$, from Lemma 3.1, the following inequality holds:

$$DIST(r, NN(r, N)) \leq NXNDIST(M, N) \tag{3.7}$$

This produces a contradiction to inequality (3.6). □

Lemma 3.4 ensures the correctness of the traversal algorithms and pruning heuristics presented in Section 3.3.2.

***Lemma* 3.5.** *Let m be a child MBR of M, and let n be a child MBR of N, then* $MINMINDIST(m, n)$ *is not always smaller than* $NXNDIST(M, N)$.

Proof. Suppose that the following inequality always holds:

$$MINMINDIST(m, n) < NXNDIST(M, N) \tag{3.8}$$

Algorithm 3.1: $NXNDIST(M, N)$

```
1  MAXDIST[D] ⇐ [0], MAXMIN[D] ⇐ [0];
2  S ⇐ 0, minS ⇐ 0;
3  for d = 1 to D do
4  |   MAXDIST[d] ⇐
   |   max(|M.L[d] − N.U[d]|, |M.L[d] − N.L[d]|, |M.U[d] − N.U[d]|, |M.U[d] − N.L[d]|);
5  |   S+ = MAXDIST[d]^2;
6  minS ⇐ S;
7  for d = 1 to D do
8  |   MAXMIN[d] ⇐ MAXMIN(M.L[d], M.U[d], N.L[d], N.U[d]);
9  |   minS ⇐ min(minS, S − MAXDIST[d]^2 + MAXMIN[d]^2);
10 return √minS;
```

We construct a counter example in Figure 3.5 to contradict this claim. As shown in the figure, $m \subset M$ and $n \subset N$. Simple distance calculations show that $NXNDIST(M, N) = \sqrt{74}$, and $MINMINDIST(m, n) = \sqrt{89}$. This produces a contradiction to inequality 3.8. □

Lemma 3.5 presents an important property of the NXNDIST that makes it a more efficient upper bound for pruning than the MAXMAXDIST metric.

3.3.1.2 Computing NXNDIST

Since NXNDIST is computed frequently during the evaluation of ANN, it is crucial to have an efficient algorithm for computing it. From Definition 3.5 we have developed an $O(D)$ algorithm for computing NXNDIST, which is shown in Algorithm 3.1.

As is shown in Algorithm 3.1, the MBRs M and N are represented by two vectors, each of size D, indicating the lower and upper bounds in each of the D dimensions. The lower and upper bounds of M in dimension d are accessible via $M.L[d]$ and $M.U[d]$ respectively. The same also applies to N.

Algorithm 3.1 proceeds in two iterations: the first iteration accumulates $S = \sum_{d=1}^{D} MAXDIST^2[d]$; the second iteration computes the $MAXMIN[d]$ value in each dimension d and obtains

Algorithm 3.2: $MAXMIN(l_r, u_r, l_s, u_s)$

```
1  MAXMIN ⇐ ∞;
2  diff_ll ⇐ l_r − l_s, diff_lu ⇐ l_r − u_s;
3  diff_ul ⇐ u_r − l_s, diff_uu ⇐ u_r − u_s;
4  len_s ⇐ u_s − l_s, mid_s ⇐ (u_s + l_s)/2;
5  if diff_ul ≤ 0 then MAXMIN ⇐ |diff_ll| //r1;
6  else if diff_ll < 0 && u_r ≤ mid_s then MAXMIN ⇐ max(|diff_ll|, |diff_ul|) //r2;
7  else if diff_ll < 0 && diff_uu ≤ 0 then MAXMIN ⇐ max(|diff_ll|, len_s/2) //r3;
8  else if diff_ll < 0 && diff_uu > 0 then MAXMIN ⇐ max(len_s/2, |diff_ll|, |diff_uu|) //r4;
9  else if diff_lu ≥ 0 then MAXMIN ⇐ diff_uu //r8;
10 else if diff_ll ≥ 0 && diff_uu ≤ 0 then MAXMIN ⇐ min(len_s/2, |diff_lu|, |diff_ul|) //r5;
11 else if l_r ≥ mid_s && diff_uu > 0 then MAXMIN ⇐ max(|diff_lu|, |diff_uu|) //r7;
12 else if diff_ll ≥ 0 && diff_uu > 0 then MAXMIN ⇐ max(len_s/2, |diff_uu|) //r6;
13 return MAXMIN
```

$$
\begin{aligned}
NXNDIST(M,N) &= \sqrt{\min_{d=1,\dots,D}\left(S - MAXDIST^2[d] + MAXMIN^2[d]\right)} \\
&= \sqrt{S - \max_{d=1,\dots,D}\left(MAXDIST^2[d] - MAXMIN^2[d]\right)}
\end{aligned}
$$

Figure 3.2 shows a 3-D example of Algorithm 3.1.

The MAXMIN procedure for producing the MAXMIN value in each dimension is shown in Algorithm 3.2. Figure 3.6 enumerates the eight cases that must be considered during the computation of MAXMIN value in dimension d. In this algorithm, s indicates the interval that is bounded by the lower and upper bounds of the projected interval of N in dimension d, whereas $r_i (i = 1, 2, ..., 8)$ indicates the possible positions of the bounded interval of M's projection in dimension d relative to that of N.

The MAXMIN algorithm takes four parameters, namely: l_r, the lower bound of the projected M interval in dimension d; u_r, the upper bound of the projected M interval in dimension d; l_s, the lower bound of the projected N interval in dimension d; u_s, the upper bound of the projected N interval in dimension d. Corresponding cases are indicated in the comments of the procedure presented in Algorithm 3.2.

3.3.2 The ANN Algorithms

3.3.2.1 Data Structures

Before presenting the actual ANN algorithms, we briefly describe two data structures that are used by these algorithms.

The first data structure, which is crucial to the pruning heuristics, is the Local Priority Queue (*LPQ*). During the ANN procedure, each entry within I_R becomes the *owner* of exactly one *LPQ*, in which a priority queue stores entries from I_S. Each entry e within the priority queue keeps a MIND and a MAXD field, accessible as e.MIND and e.MAXD, respectively. These fields indicate the lower bound and upper bound of the distance from the *owner*'s MBR to e's MBR.

The entries in the priority queues inside the *LPQ*s are ordered by their MIND field. In addition, each *LPQ* also keeps a MAXD field which records the minimum (for ANN queries) or maximum (for AkNN queries) of all e.MAXD values in the priority queue, as the upper bound for pruning unnecessary entries.

There are two reasons for using *LPQ*: (i) By requiring the *owner* of each of the *LPQ*s to be unique, we avoid duplicate node expansions from I_R (thus improving beyond the bitmap approach of [29, 46], since the bitmap approach only builds a bitmap for the point data objects within **R**, but not the intermediate node entries); (ii) *LPQ* gives us the advantages of the Three-Stage pruning heuristics, which we discuss in detail in Section 3.3.2.7.

The second data structure is simply a FIFO Queue, which serves as a container for the *LPQ*s during node expansions.

3.3.2.2 The Top Level ANN Procedure

The top level ANN procedure, which is common to all our ANN algorithms, is shown in Algorithm 3.3. The key part of this algorithm is calling the *ExpandAndPrune* method, which performs a bi-directional expansion of the entries in the root nodes of I_R and I_S.

Algorithm 3.3: $ANN(I_R, I_S, algo, Result)$

```
1  Q_root ⇐ NewFIFOQueue();
2  LPQ_root ⇐ NULL;
3  LPQ_root.owner ⇐ I_R.root;
4  Distances(LPQ_root.owner, I_S.root);
5  LPQ_root.push(I_S.root);
6  ExpandAndPrune(BI, LPQ_root, Q_root, Result);
7  if algo =BFBI then ANN-BFBI(Q_root, Result);
8  else if algo =BFUNI then ANN-BFUNI(Q_root, Result);
9  else if algo =DFBI then
10 |   while LPQ_new ⇐ dequeue(Q_root) do  ANN-DFBI(LPQ_new, Result);
11 else if algo =DFUNI then
12 |   while LPQ_new ⇐ dequeue(Q_root) do  ANN-DFUNI(LPQ_new, Result);
```

For each entry in I_R, a *LPQ* is constructed, which is populated by entries from I_S (and the entry for the owner of the *LPQ*). The top level FIFO Queue essentially contains all the *LPQ*s that are built. (The *ExpandAndPrune* procedure is described in more detail in Section 3.3.2.7.) After this initialization, the search algorithm specified by the *algo* parameter is invoked. In the following sections we give brief descriptions on each of the four ANN algorithms.

3.3.2.3 The ANN-BFBI Algorithm

Algorithm 3.4 outlines the ANN-BFBI algorithm which employs a Breadth-First traversal of I_R, with BI-directional node expansion. The procedure *ExpandAndPrune*, detailed in Algorithm 3.8, essentially expands an entry either in a uni-directional or a bi-directional way (which is controlled by the first argument), and applies a pruning technique to limit the number of expanded entries that need to be considered further.

The ANN-BFBI algorithm traverses the index on I_R iteratively. The traversal on I_R is achieved level by level, with a FIFO Queue (Q_{in} and Q_{out} in Algorithm 3.4) constructed and populated with the *LPQ*s (*LPQ* in Algorithm 3.4) *owned* by all entries from I_R on that particular level. With bi-directional node expansion, I_S is explored synchronously with I_R. Since entries at each level in both I_R and I_S are visited only once, ANN-BFBI is very I/O

Algorithm 3.4: $ANN-BFBI(Q_{in}, Result)$

```
1 if Q_in not empty then Q_out ⇐ ∅;
2 while Q_in not empty do
3     LPQ ⇐ dequeue(Q_in);
4     ExpandAndPrune(BI, LPQ, Q_out, Result);
5 ANN-BFBI(Q_out, Result);
```

Algorithm 3.5: $ANN-BFUNI(Q_{in}, Result)$

```
1 if Q_in not empty then Q_out ⇐ ∅;
2 while Q_in not empty do
3     LPQ ⇐ dequeue(Q_in);
4     ExpandAndPrune(UNI, LPQ, Q_out, Result);
5 ANN-BFUNI(Q_out, Result);
```

efficient.

3.3.2.4 The ANN-BFUNI Algorithm

The ANN-BFUNI algorithm (Breadth-First traversal of I_R, with UNI-directional node expansion) is shown in Algorithm 3.5. Similar to ANN-BFBI, I_R is traversed in a level-by-level fashion, with one FIFO Queue for each level. Lower level FIFO Queues are derived from higher level ones by dequeuing *LPQ*s from them and expanding the entries uni-directionally.

3.3.2.5 The ANN-DFBI Algorithm

Algorithm 3.6 shows the ANN-DFBI algorithm: Depth-First traversal of I_R, with BI-directional node expansion. In this algorithm, the index I_R is explored in a depth-first fashion. As a result, the FIFO Queue (Q_{out} in Algorithm 3.6) at each level will only contain *LPQ*s (LPQ_{child} in Algorithm 3.6) obtained by expanding both the *owner* entry of the higher level *LPQ* (LPQ_{in} in Algorithm 3.6) and the entries residing inside the priority queue contained within that *LPQ*. Consequently, memory consumption is dramatically reduced compared to ANN-BFBI. In addition, because bi-directional node expansion implies synchronous traversal of both indexes, data locality is also maximized, which

Algorithm 3.6: $ANN-DFBI(LPQ_{in}, Result)$

```
1 Q_out ⇐ ∅ ;
2 ExpandAndPrune(BI, LPQ_in, Q_out, Result);
3 while Q_out not empty do
4     LPQ_child ⇐ dequeue(Q_out);
5     ANN-DFBI(LPQ_child, Result);
```

Algorithm 3.7: $ANN-DFUNI(LPQ_{in}, Result)$

```
1 Q_out ⇐ ∅;
2 ExpandAndPrune(UNI, LPQ_in, Q_out, Result);
3 while Q_out not empty do
4     LPQ_child ⇐ dequeue(Q_out);
5     ANN-DFUNI(LPQ_child, Result);
```

improves I/O efficiency.

3.3.2.6 The ANN-DFUNI Algorithm

Depth-First traversal of I_R, with UNI-directional node expansion, employs the same index traversal technique as ANN-DFBI and is presented in Algorithm 3.7. The uni-directional node expansion approach makes ANN-DFUNI essentially the same as the naive approach, where a NN query on I_S is issued for each point data object within I_R.

3.3.2.7 Pruning Heuristics

In this section, we discuss the *ExpandAndPrune* algorithm, which is presented in Algorithm 3.8.

The basic heuristic for pruning is as follows: Let PM represent the chosen pruning metric between two MBRs M and N (PM could be MAXMAXDIST or NXNDIST). The pruning rule is that if $MINMINDIST(M, N) > PM(M, N')$, for some N', then the path corresponding to (M, N) can be safely pruned.

The LPQ owned by each unique entry on I_R acts as the main filter, and enforces three stages of pruning: Expand Stage, Filter Stage, and Gather Stage.

The Expand Stage refers to the stage when *owner*s of LPQs are internal nodes on I_R

Algorithm 3.8: ExpandAndPrune(dir, LPQ_{in}, Q_{out}, Result)

```
1  if owner of LPQ_in is an OBJECT then
2      initialize LPQ_n and set its owner to that of LPQ_in;
3      pass MAXD of LPQ_in to LPQ_n;
4      popped ⇐ 0;
5      while n ⇐ LPQ_in.pop() do
6          popped ⇐ popped + 1;
7          if n is an OBJECT &&popped = 1 then
8              Result.push(LPQ.owner, n);
9              break;
10         else
11             forall e ∈ n do
12                 Distances(LPQ_n.owner, e);
13                 if e.MIND ≤ LPQ_n.MAXD then LPQ_n.push(e);
14     push LPQ_n into Q_out;
15 else
16     forall c ∈ LPQ_in.owner do
17         initialize LPQ_c and set its owner to c;
18         pass MAXD of LPQ_in to LPQ_c;
19     while n ⇐ LPQ_in.pop() do
20         if dir = BI then
21             forall e ∈ n do
22                 forall c ∈ LPQ_in.owner do
23                     Distances(LPQ_c.owner, e);
24                     if e.MIND ≤ LPQ_c.MAXD then LPQ_c.push(e);
25         else if dir = UNI then
26             forall c ∈ LPQ_in.owner do
27                 Distances(LPQ_c.owner, n);
28                 if n.MIND ≤ LPQ_c.MAXD then LPQ_c.push(n);
29     push all non-empty LPQ_c into Q_out;
```

and are expanded, new lower level *LPQ*s (LPQ_c as shown in Algorithm 3.8) are created for and owned by their child entries (c in Algorithm 3.8). At this time, the MAXD field from the old *LPQ* (LPQ_{in} parameter in Algorithm 3.8) is passed on to the new *LPQ*s, and is used as the initial pruning upper bound (Lines 2-3, 18-19 in Algorithm 3.8). As entries (n in Algorithm 3.8) are popped out of LPQ_{in}, their MIND field, which holds the MINMINDIST value from the MBR of n to that of $LPQ_{in}.owner$ is compared against

$LPQ_c.MAXD$, and if it's smaller, n is further processed. At this time, depending on the value of the *dir* parameter, two cases apply.

Case 1 ($dir = BI$): In this case, n is expanded; its children are probed against all LPQ_cs, their MIND and MAXD values are computed against the owners of the LPQ_cs (inside the *Distances* function in Algorithm 3.8), and are compared against the MAXD fields of the LPQ_cs. At this time, these child entries are either discarded or queued by the LPQ_c's, and if so, updating their MAXD fields, respectively (Lines 21-24). In this case NXNDIST has additional pruning advantages over MAXMAXDIST due to Lemma 3.5, namely, early pruning becomes possible even when the MAXD field of the LPQ_cs has not yet been updated, which is not possible when MAXMAXDIST is used.

Case 2 ($dir = UNI$): This is the uni-directional node expansion case. n is not expanded, but instead, its MIND and MAXD are re-computed against the owners of the LPQ_cs using the *Distances* function, and is either discarded or queued by the LPQ_cs, and if so, updating their MAXD fields, respectively (Lines 27-29).

Note in the Expand Stage, the pruning happens in three places: when n is first popped out of LPQ_{in}; when entries(either n themselves, or n's child entries) are probed against the LPQ_cs; and when the MAXDs of the LPQ_cs are updated, i.e., reduced by previously queued entries, the n's that come in later will see a much tighter upper bound.

The Filter Stage happens in the *push* function in Algorithm 3.8. We observe that it is possible that during the Expand Stage, the MAXD of a new incoming entry may become smaller than the MIND of some entries that are already inside the queue, just because those entries were pushed into the queue earlier, when the MAXD field of the corresponding *LPQ* was not yet updated. This may lead to serious performance degradation since more nodes than necessary will be expanded/explored in the next iteration. To address this problem, we activate the Filter Stage.

During the Filter Stage, as the new node is being pushed into the priority queue inside a *LPQ*, its MAXD is compared against the MIND field of all the entries that it passes as

it floats up to the top of the queue, searching for entries with a MIND that is larger than its own MAXD. When such an entry is found, it is replaced by the new entry, instead of being swapped down to a lower level of the priority queue. During this stage, there may be a tie on the MIND value. We break the tie by comparing the the MAXD fields of these two entries, and swap the new entry with the old one if the new MAXD is less than the old entry's. In doing so, we are essentially optimizing the locality of pruning heuristics.

The Gather Stage refers to the stage when the *owner* of LPQ_{in} is an actual point data object, then as entries are popped out of LPQ_{in}, if the first out is also an actual data object, then the search is over for this particular object (Lines 7-9). Otherwise, the entry is expanded and processed, updating the MAXD field of LPQ_n accordingly (Lines 11-16).

Note that the Three-Stage-Pruning strategy proposed here is a general-case optimization technique for ANN processing and can be easily adapted on any indices where the upper bound is non-increasing during the search.

3.3.2.8 Effectiveness of NXNDIST

The Three-Stage-Pruning strategy discussed above becomes extremely effective when NXNDIST is used as the upper bound for pruning, compared to MAXMAXDIST. The reasons for this effect are as follows: (a) NXNDIST by itself is a much tighter upper bound than MAXMAXDIST, so the chances of the NXNDIST of a new entry being less than the MIND field of an existing entry in the priority queue become much higher. (b) As the search descends down the indices, the reduction in the length of NXNDIST is higher than that of MAXMAXDIST (see Lemma 3.5). As a result, better pruning is achieved with NXNDIST as it discards non-leaf nodes that don't need to be expanded – which drastically reduces the number of the next level nodes to examine.

3.3.3 Extension to AkNN

The intuition behind the extension of our method to compute All-*K*-Nearest-Neighbor (AkNN) is as follows: At any time, in order to guarantee *k* NN results for all point objects

Algorithm 3.9: AkNN_ExpandAndPrune(dir, LPQ_{in}, Q_{out}, Result)

```
1  if owner of LPQ_in is an OBJECT then
2      initialize LPQ_n and set its owner to that of LPQ_in;
3      pass MAXD of LPQ_in to LPQ_n;
4      popped ⇐ 0;
5      while n ⇐ LPQ_in.pop() do
6          if n is an OBJECT then
7              Result.push(LPQ.owner, n);
8              if popped = k then break;
9              else popped ⇐ popped + 1;
10         else
11             forall e ∈ n do
12                 Distances(LPQ_n.owner, e);
13                 if e.MIND≤ LPQ_n.MAXD||LPQ_n.size < k then LPQ_n.push(e);
14     push LPQ_n into Q_out;
15 else
16     forall c ∈ LPQ_in.owner do
17         initialize LPQ_c and set its owner to c;
18         pass MAXD of LPQ_in to LPQ_c;
19     while n ⇐ LPQ_in.pop() do
20         if dir = BI then
21             forall e ∈ n do
22                 forall c ∈ LPQ_in.owner do
23                     Distances(LPQ_c.owner, e);
24                     if e.MIND≤ LPQ_c.MAXD||LPQ_c.size < k then LPQ_c.push(e);
25         else if dir = UNI then
26             forall c ∈ LPQ_in.owner do
27                 Distances(LPQ_c.owner, n);
28                 if n.MIND≤ LPQ_c.MAXD||LPQc.size < k then LPQ_c.push(n);
29     push all non-empty LPQ_c into Q_out;
```

within the *owner* of a *LPQ*, there must be at least k entries from I_S in the *LPQ*. An entry e from I_S can only be pruned away when there are at least k entries in the *LPQ* and the MINMINDIST from the *owner* MBR to that of e is greater than the MAXD field of the *LPQ*.

The extension of our methods to AkNN processing [18, 112] can be realized through slight modifications of the *ExpandAndPrune* algorithm (Algorithm 3.8), using NXNDIST

and the parameter k as the pruning metric. The modified *AkNN_ExpandAndPrune* algorithm is shown in Algorithm 3.9. Notice the only parts that have changed from Algorithm 3.8 are the termination condition (lines $7-11$), and the filtering conditions (lines 15, 27, and 32). These changes incorporate the additional cardinality constraint of the *LPQ*s. The pruning heuristics discussed in Section 3.3.2.7 are still applicable to AkNN, with fairly straight-forward modifications. (In the interest of space we omit these extensions here.)

3.3.4 MBRQT

The ANN algorithms and NXNDIST metric proposed so far are both general purpose and can be applied to various index structures that incorporate the notion of MBR and Euclidean distance metrics. In a number of previous ANN works [28, 29, 46, 95, 116], the R*-tree index has been used. This is understandable since R*-tree is the "ubiquitous" spatial indexing structure. However it is natural to ask if other indexing structures have an advantage over the R*-tree for ANN processing. Notice that the R*-tree family of indices basically partition the underlying space based on the actual data distributions. Consequently, the partition boundaries for two R*-trees on two different datasets will be different. As a result when running ANN, the effectiveness of the pruning metrics such as NXNDIST will be reduced, as the pruning heuristic relies on this metric being smaller than some MINMINDIST. In contrast, an indexing method that imposes a regular partitioning of the underlying space is likely to be much more amenable to the pruning heuristic. A natural candidate for a regular decomposition method is the quadtree [93]. We do note that quadtrees are not a balanced data structure, but they can be mapped to disk resident structures quite effectively [33, 47], and some commercial DBMSs already support quadtrees [60]. The question that we raise, and answer, in this chapter is how effective is a quadtree index compared to an R*-tree index for ANN processing.

Note that with a traditional quadtree, spatially neighboring nodes all border each

other and the pairwise MINMINDIST value is zero. This may inevitably cause excessive memory overhead due to large queue or stack size resulting from a low pruning rate. To fix this this problem, we associate an explicit MBR with each internal node, which produces a tighter approximation of the entries below that node (though at the cost of increasing storage cost). Essentially, our proposal is to enhance a regular PR bucket quadtree with MBRs. This enhanced indexing structure is called the MBR-quadtree, or simply MBRQT. As our experimental results show this index structure is significantly more effective than R*-trees for ANN processing.

3.4 Experimental Evaluation

In this section, we present the results of our experimental evaluation. We first evaluate the effectiveness of the various ANN algorithms proposed in Section 3.3, using both the MBRQT index structure and the R*-tree index structure, with NXNDIST as the pruning metric.

Then, we compare our ANN methods with previous ANN algorithms. Of all the previously proposed ANN methods, the recent batch NN (BNN) [116] and GORDER [112] methods are considered to be the most efficient. Consequently, in our empirical evaluations, we only compare our method with these two methods.

We note that BNN and GORDER haven't actually been compared to each other in previous work. A part of the contribution that we make via our experimental evaluation is to also evaluate the relative performance of these two methods.

3.4.1 Implementation Details

We have implemented a persistent MBRQT and an R*-tree on top of the SHORE storage manager [22]. We compiled the storage manager with 8KB page size, and set the buffer pool size to 64 pages ($512KB$). The purpose of having a relatively small buffer pool size is to to keep the experiments manageable, which also essentially follows the

experimental design philosophy used in previous research [64, 71, 105, 116]. At these smaller buffer pool sizes, even with small datasets, we can easily see the breakdown of the IO and the CPU costs.

We have also experimented with various buffer pool sizes, and the conclusions presented in this section hold even for these larger buffer pool sizes. In the interest of space, these additional experiments are suppressed in this presentation. One exception to this behavior, is the performance of GORDER, which is very sensitive to the buffer pool size for high-dimensional dataset. To quantify this effect, we present one experiment with varying buffer pool sizes (in Section 3.4.5).

For both MBRQT and the R*-tree the leaf node size is set to the storage manager page size, and the non-leaf nodes in MBRQT are simply small objects. We do not employ any specific packing strategy for the MBRQT non-leaf nodes, but simply use the default clustering mechanism provided by the storage manager.

For the set of experiments that compare the MBRQT approach against previous methods, we take advantage of the original source code generously provided by the authors of [116] and the authors of [112]. For consistency, we modified the BNN implementation, switched the default page size from 4KB to 8KB, and retained the LRU cache size of 512KB. The parameters used for the GORDER methods are chosen using the suggested optimal values in the experimental section of [112], and K is set to 1 for all of the experiments comparing the ANN performance of these methods.

All experiments were run on a 1.2GHz Intel Pentium M processor, with 1GB of RAM, running Red Hat Linux Fedora Core 2. For each measurement that we report, we actually ran the experiment five times. We then took the average of the middle three numbers, and report this number.

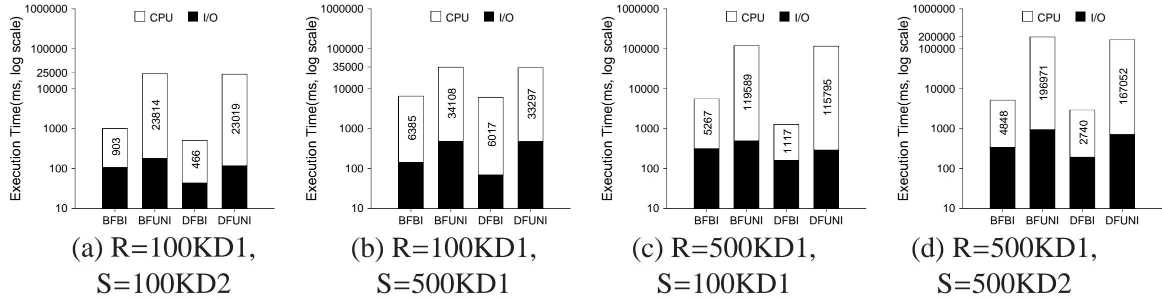
Table 3.2: Table of Experimental Datasets

Dataset	Cardinality	Dimensions	Description
100K2D	100K	2	2D 100K point data
500K2D	500K	2	2D 500K point data
500K4D	500K	4	4D 500K point data
500K6D	500K	6	6D 500K point data
TAC	700K	2	Twin Astrographic Catalog Data
FC	580K	10	Forest Cover Type data

3.4.2 Experimental Datasets and Workload

We perform experiments on both real and synthetic datasets. We use two real datasets: The Twin Astrographic Catalog dataset (TAC) from the U.S. Naval Observatory site [3], and the Forest Cover Type (FC) from the UCI KDD data repository [2]. The TAC data contains high quality positions of around 700K stars. This dataset is a 2D dataset. The Forest Cover dataset contains information about various 30 x 30 meter cells for the Rocky Mountain Region (US Forest Service Region 2). Each tuple in this dataset has 54 attributes, of which 10 attributes are real numbers. The ANN operation is run on these 10 attributes (following similar use of this dataset in previous ANN works, such as [112]).

We also modified the popular GSTD data generator [107] to produce medium-to-large scale multi-dimensional synthetic datasets. We produced synthetic datasets by varying the number of objects from $100K$ to $500K$. Although we experimented with various combinations of datasets with a wide range of sizes, in the interest of space, we only present selected results from a few representative workloads. The synthetic datasets that we use in this section are: $100K$ object points to represent relatively small datasets, and $500K$ object points to represent large sized datasets. To test the effect of data dimensionality on the ANN methods, two more datasets of cardinality 500K are also generated, with dimensionality of 4 and 6, respectively. Table 3.2 summarizes the datasets that we use in our experiments.



(a) R=100KD1, S=100KD2 (b) R=100KD1, S=500KD1 (c) R=500KD1, S=100KD1 (d) R=500KD1, S=500KD2

Figure 3.7: Evaluating the ANN Algorithms

3.4.3 Evaluating the ANN Algorithms

In this experiment, we compare the performance of the family of four ANN algorithms presented in Section 3.3.2. In the interest of space, we only present the results with the synthetic datasets.

Figure 3.7 summarizes the results for the four ANN algorithms using the MBRQT indexing method. All graphs in this figure show both the IO and the CPU components in the total query execution cost. All the y-axis in these graphs use a log scale. Further, the number in the CPU portion of the bar shows the actual CPU times.

Effect of bi-directional v/s uni-directional expansion: From Figure 3.7, we observe that with the traversal pattern fixed, the CPU cost for the bi-directional node expansion technique is lower than that for the uni-directional technique by at least an order of magnitude. This is because with bi-directional node expansion, new nodes are produced at a quadratic rate, and if an efficient pruning metric such as NXNDIST is used, a lot of early pruning occurs at the non-leaf node levels. This behavior further magnifies the effect of pruning, resulting in much smaller number of distance calculations than if uni-directional expansion technique is used. The reduction in number of nodes that have to be considered (because of better pruning), also leads to a lower IO cost for the bi-directional method.

Effect of depth-first v/s breadth-first traversal: From Figure 3.7, we observe that with a fixed node expansion technique, the depth-first traversal technique outperforms breadth-first traversal. The depth-first method has both lower IO and CPU costs. With

a breadth-first traversal the index entries are expanded level-by-level, which results in repeated accesses to index entries. Since these accesses are spread across the entire run of the algorithm, it resulted many more random IOs.

The depth-first method also has lower CPU costs. This is because a depth-first expansion will quickly result in examining index entries that are deeper down in the tree. These entries have smaller MBRs and result in more accurate NXNDIST values. Consequently, there is better pruning with the depth-first method, which results in a lower CPU cost. In addition, the depth-first method has a much smaller memory footprint.

To summarize the results shown in Figure 3.7, we note that ANN-DFBI is the most efficient of the four ANN algorithms with respect to both CPU and I/O performance. The ANN-BFBI is the second best method, and the ANN-DFUNI algorithm has the lowest performance.

We also repeated this same set of experiments using the R*-tree index. The results we obtained were consistent with the conclusion that ANN-DFBI is consistently the most efficient alternative. In the interest of space, these results are not shown here. We also observed that MBRQT consistently outperforms the R*-tree method across all four ANN methods. In the interest of space we omit these results here, but present the comparison using the TAC dataset in Figure 3.8. In this figure the bars corresponding to "RBA NXNDIST" and "MBA NXNDIST" present a direct comparison of the ANN-DBFI method with the two index structures. As can be seen in this figure, *simply switching the indexing structure from R*-tree to MBRQT improves the overall performance of ANN search by 3X*, for reasons discussed in Section 3.3.4.

For the remainder of this section, we only consider the ANN-DFBI algorithm. For simplicity, we refer to this ANN method as *MBA* (**M**BRQT **B**ased **A**NN method) and *RBA* (**R***-tree **B**ased **A**NN method) for the implementation with the MBRQT and the R*-tree indexing methods respectively.

3.4.4 Effectiveness of the NXNDIST Metric

In this experiment, we evaluate the effectiveness of the NXNDIST metric and compare it with the traditional, looser pruning metric – MAXMAXDIST. For this experiment, we use the TAC dataset. Since BNN [116] is currently the most efficient R*-tree based ANN method, we compare both our MBA and RBA methods with BNN. The results of this experiment are shown in Figure 3.8.

In Figure 3.8, results for, BNN, MBA, and RBA approaches are shown, with both the MAXMAXDIST and the new NXNDIST pruning metric. (Similar results are also observed with the synthetic datasets, which we omit here in the interest of space.) Note that the original BNN algorithm of [116] corresponds to the bars labeled as "BNN MAXMAXDIST", and the BNN algorithm with NXNDIST as the pruning metric corresponds to the bars labeled as "BNN NXNDIST".

An informed reader may note that the original BNN algorithm has a *globaldist* parameter, which is set to some *MAXREAL* that is defined in the code. We replaced this with MAXMAXDIST and have observed that it improves the performance slightly over the original version. Similarly, setting the *globaldist* parameter to NXNDIST gives us the BNN NXNDIST algorithm.

From Figure 3.8, we notice that for all three methods, BNN, MBA, and RBA, the use of NXNDIST metric dramatically improves the query performance. *Observe the order-of-magnitude improvement in execution time for the MBA method, and a 6X performance gain for both the BNN and RBA methods, by simply switching to the new NXNDIST metric*.

The drastic improvement of NXNDIST over MAXMAXDIST is due to reasons discussed in Section 3.3.2.8. Also, the slightly reduced effect of NXNDIST on BNN and RBA can be attributed to the MBR overlapping problem inherent with R*-trees (see Section 3.3.4), which reduces the effectiveness of the pruning metrics. For example, for a certain MBR in I_R, overlapping MBRs within I_S often have very similar lower and upper

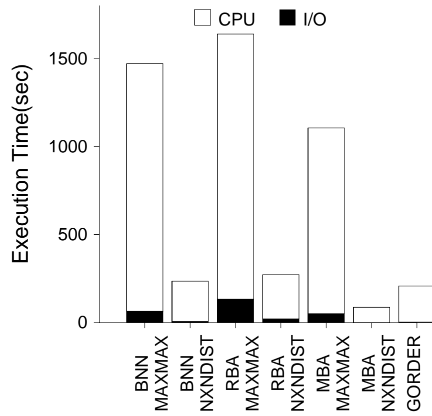


Figure 3.8: Comparison of Index Structures and Methods: TAC Data(2D)

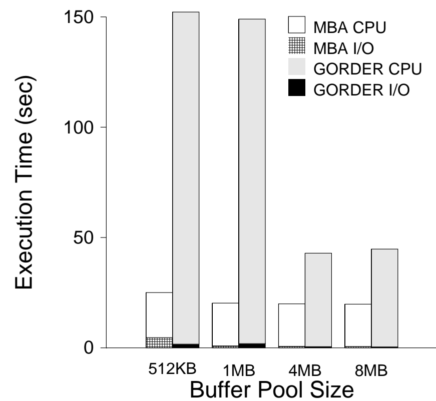


Figure 3.9: Comparison of Index Structures and Methods: FC Data(10D)

distance bounds, and thus become harder to prune.

3.4.5 Comparison of BNN, MBA, and GORDER

In Figures 3.8 and 3.9 we show the results comparing BNN, MBA, and GORDER using the two real datasets.

BNN v/s MBA: For this comparison, consider Figure 3.8. Comparing BNN and MBA in this figure, we observe that with the same pruning metric, *MBA is superior to the R*-tree BNN algorithm, both in terms of the CPU cost and the I/O cost*. The superior performance of MBA over BNN is a result of the underlying MBRQT index, which has the advantages of the regular non-overlapping decomposition strategy employed by the quadtree (see Section 3.3.4 for details).

GORDER v/s BNN: From Figure 3.8 we observe that in general the *GORDER algorithm is superior to the BNN method*. There are two main reasons: (a) Both methods employ techniques to group the datasets to maximize locality. However, BNN does this only for the dataset **R**, while in GORDER the locality optimization is achieved by partitioning both input datasets and by using a transformation to produce nearly uniform datasets. (b) In BNN, an R*-tree index is built for dataset **S**. The inherent problem of overlapping MBRs in an R*-tree results in both higher I/O and CPU costs during the index

traversal. In GORDER, however, the two datasets are disjointly partitioned, which leads to better CPU and I/O characteristics.

We also compared GORDER and BNN for the synthetic datasets, and found that GORDER was faster than BNN in all cases (these results have been suppressed in the interest of space). Since GORDER is faster than BNN, for the remainder of this section we only present results comparing our MBA method with GORDER.

GORDER v/s MBA: The results in Figure 3.8 show that *MBA outperforms GORDER by at least 2X* on the two-dimensional TAC dataset. The reasons for these performance gains are three-fold: (a) GORDER requires repeated retrievals of the dataset **S**, while MBA traverses the indices I_R and I_S simultaneously. This synchronized traversal of the indices results in better locality of access, which results in fewer buffer misses; (b) The pruning metric employed in GORDER is similar to that in BNN, initially set to a certain sentinel value (the *MAXREAL* value, described in Section 3.4.4). Although this value is updated as the algorithm proceeds, it is set using the MAXMAXDIST metric, which is less effective than the NXNDIST (as discussed in Section 3.4.4); (c) The MBRQT index structure of MBA has an advantage over the nested-loops join algorithm employed by GORDER. With MBRQT, the pruning happens at multiple levels of the index structure, where early non-leaf node level pruning will save a significant amount of computation. GORDER, on the other hand, is essentially a block nested-loops join algorithm, with the pruning happening only on the block and object levels, and thus incurring significantly more distance computations.

The *performance advantages of MBA over GORDER continue for higher dimensional datasets*. Figure 3.9 shows the execution time for these two algorithms on the 10-dimensional FC dataset. We also use this experiment to illustrate the effect of buffer pool size on the GORDER method when using high-dimensional datasets[1]. To quantify this

---

[1]We note that the performance of GORDER is sensitive to the buffer pool size only for high-dimensional datasets. For low-dimensional datasets the buffer pool effects are very small. For example, with the TAC data

effect, for this experiment, we vary the buffer pool size from 512KB to 8MB.

The first observation to make in Figure 3.9 is the performance of GORDER improves rapidly as the buffer pool size increases from 1MB to 4MB, after the 4MB point the performance of GORDER is stable. The reason for this behavior with GORDER is as follows: GORDER essentially executes a block nested loops join and is joining a single block of the outer relation R with a number of blocks of the inner relation S. Before actually executing an in-memory join of the data in "matching" R and S blocks, GORDER uses a distance based pruning criteria to safely discard pairs of blocks that are guaranteed to not produce any matches. This distance pruning is more effective when there are larger number of S blocks to examine, which happens naturally at larger buffer pool sizes. Since the pruning criteria is influenced by the number of neighbors of a grid cell (which grows rapidly as the dimensionality increases), the effect of the smaller buffer pool size is more pronounced at higher dimensions. On the other hand, as discussed in Section 3.3.2.5, the MBA algorithm using MBRQT only keeps a small number of candidate entries from I_S, inside the LPQ for each R index entry. Spatial locality is thus preserved and the performance is not significantly affected by the size of the buffer pool.

The second observation to make in Figure 3.9 is that MBA is consistently faster than GORDER for all buffer pool sizes. For larger buffer pool sizes MBA is 2X faster, and for smaller buffer pool sizes it is 6X faster.

3.4.6 Effect of Dimensionality

In this section, we systematically increase the data dimensionality, and measure its effect on the performance of MBA and GORDER.

For this experiment, we generated a number of synthetic datasets, with varying cardinalities and dimensionalities. In the interest of space we show in Figure 3.10 results for a representative workload, namely the 500K2D, 500K4D, and 500K6D datasets. (The

changing the buffer pool size from 512KB to 8MB only improved the performance of GORDER by 5%.

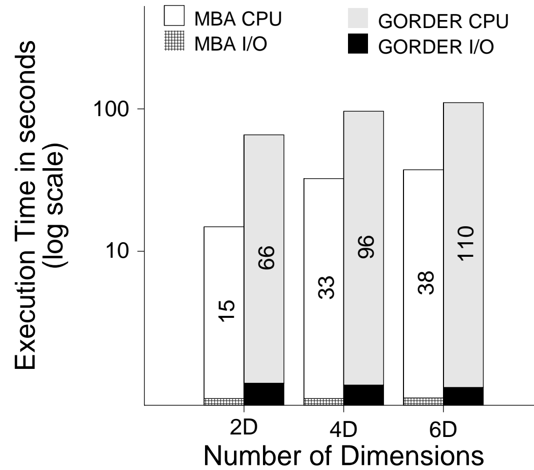


Figure 3.10: Effect of Dimensionality

numbers in the bars in this graph show the actual CPU costs in seconds.)

As is shown in the figure, *MBA consistently outperforms GORDER by approximately 3X for all 2D, 4D, and 6D datasets*. As the dimensionality of the data increases, the CPU time for both methods increases very gradually, and the I/O time also elegantly scales up with the dimensionality of the datasets. This observation is consistent for both the TAC and FC datasets in Figures 3.8 and 3.9.

As we have noted previously, ANN is a very computationally intensive operation, and most of the execution time is spent on distance computation and comparisons. Thus, having an efficient distance computation algorithm for high-dimensional data is crucial to the performance of ANN methods. Looking at the CPU time for MBA (which uses the NXNDIST metric) in Figure 3.10, we observe that the CPU cost is not shooting up sharply as the dimensionality increases, which shows the effectiveness of the $O(D)$ NXNDIST computation algorithm (presented in Section 3.3.1.2).

3.4.7 Evaluating AkNN Performace

We use both real-world datasets, TAC and FC, for the experiment comparing AkNN performance of MBA against GORDER. We follow the example in [112] and vary k value from 10 to 50, with increment of 10. Figures 3.11 and 3.12 show the results of this

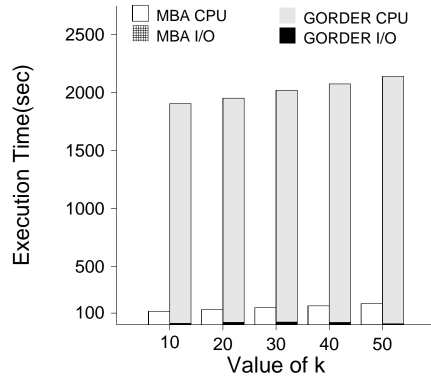


Figure 3.11: AkNN on TAC Data(2D)

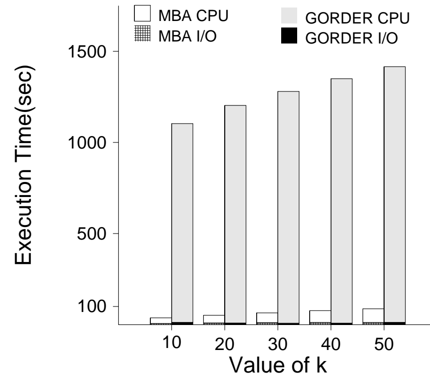


Figure 3.12: AkNN on FC Data(10D)

experiment.

As can be seen in these figures, on both the TAC and FC datasets, the execution time of MBA and GORDER increases as the k value goes up. However, *MBA is over an order of magnitude faster than GORDER in all cases*. The reasons for this performance advantage for MBA over GORDER are similar to those described in Section 3.4.5.

3.5 Conclusions

In this chapter we have presented a new metric, called NXNDIST, and have shown that this metric is much more effective for pruning ANN computation than previously proposed methods. We have also explored the properties of this metric, and have presented an efficient $O(D)$ algorithm for computing this metric, where D is the data dimensionality. We have also explored a family of index based methods for computing ANN queries. In addition, for ANN computation, we have shown that traversing the index trees using a depth-first paradigm, and using a bi-directional expansion of candidate search nodes is the most efficient strategy. With the application of NXNDIST, we have also shown how to extend our solution to efficiently answer the more general AkNN question.

Finally, we have shown that for ANN queries, using an quadtree index enhanced with MBR keys for the internal nodes, is a much more efficient indexing structure than the commonly used R*-tree index. Overall the methods that we have presented generally

result in significant speed-up of at least 2X for ANN computation, and over an order of magnitude for AkNN computation over the previous best algorithms (BNN [116] and GORDER [112]), for both low and high-dimensional datasets.

CHAPTER 4

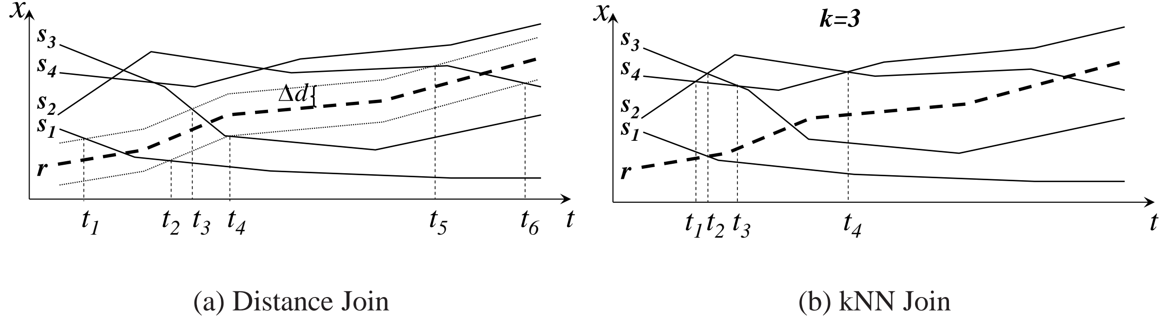
TRAJECTORY JOINS WITH APPLICATION IN PRIVACY PRESERVATION

4.1 Introduction

In Location Based Service (LBS) applications, large volumes of *trajectory* datasets are collected, where each trajectory traces the movement of a object in space and time coordinates as it moves around in physical space. A large part of previous research on trajectory processing has focused on efficient access methods for both *historical* and *predictive* trajectories (where the prediction is based on past history and follows certain motion model). While both spatial join and temporal join operations have been widely researched in the past (e.g. [18, 28, 29, 32, 68, 84, 94, 109, 112, 115–117]), very limited work has been done to address the more complex problem of trajectory join operations. We present **JiST**, a framework for trajectory join processing in spatio-temporal databases, and develop scalable algorithms for these operations.

A trajectory join, also called *spatio-temporal* join, is a join operation between two trajectory data-sets. It is defined by a combination of spatial and temporal predicates. Spatial predicates include the distance measure between trajectories (Trajectory Distance Join, or TDJ), or the number of Nearest Neighbors of certain trajectories (Trajectory *k* Nearest Neighbors Join, or TkNNJ). Temporal predicates, on the other hand, specify the duration of the join, i.e., the time window, or the temporal placement of the join, be it in the past or the future.

Figure 4.1 shows two examples of one-dimensional trajectory join operations.



(a) Distance Join

(b) kNN Join

Figure 4.1: Trajectory Joins: A One-dimensional Example

Figure 4.1(a) depicts a trajectory distance join operation, and Figure 4.1(b) gives an illustration of a TkNNJ operation. The t axis in both figures denotes the time, and the x axis shows the one-dimensional spatial extent. In both figures, there are two sets of objects, namely **R** and **S**. The objects s_i (i = 1 to 4) belong to the set **S**. For simplicity, we only show one object r in the set **R**. Thin solid lines indicate the trajectories followed by objects s_i, and thick dashed lines depict the trajectory followed by object r.

Using Figure 4.1, consider a TDJ operation, which retrieves for all objects $r \in \mathbf{R}$, all objects $s \in \mathbf{S}$, such that each s object comes within Δd distance of r at some time. For object r, this is translated into a region that is bounded by the two thin dotted lines along its trajectory. As time progresses, any object s_i whose trajectory intersects with the bounded region should be reported as part of the query result.

In Figure 4.1(b), a TkNNJ operation is shown for $k = 3$. This TkNNJ query retrieves for all objects $r \in \mathbf{R}$, their respective k nearest neighbors in dataset **S**, at any specified time instance, or within a time window. In Figure 4.1(b) the points in time when the result set changes for the query are marked on the t axis.

Trajectory join operations (TDJ or TkNNJ) have many important applications, an example of which is privacy preservation of trajectory data. To date, a number of privacy preservation techniques proposed for relational databases, such as k-Anonymity [101, 102] and l-Diversity [69], have been adapted for spatio-temporal databases [9, 34, 37, 39, 73], among which location k-Anonymity [39] has proven to be most popular and effective.

However, past methods have mainly focused on protecting privacy on current snapshot of location data, while oftentimes analysis needs to be done with privacy preservation on large amounts of historical trajectory data in repositories, on which no previous research has been done. In this chapter, we introduce the concept of *Trajectory k-Anonymity*, and devise *Trajectory Cloaking* techniques based on the JiST operations to preserve trajectory k-Anonymity.

Besides trajectory privacy preservation, there are numerous example applications on which the JiST operations can be applied in a straightforward manner. We briefly describe a few of these example applications in the following.

- During the Mad Cow Disease epidemic, in the event where some cows are identified as having contracted the disease, a decision needs to be made quickly regarding which other cows (in the same or a different herd) have come in close contact with the infected units within a certain period of time in the past, depending on how long the infected units have been sick.
- Consider a battlefield scenario where soldiers need to execute tasks in small groups at night time when limited visibility can very likely cause some soldiers to go astray. To help mitigate the damage that can potentially be caused by this problem, the central database server may periodically issue a TkNNJ operation to keep track of the nearest neighbors of each soldier and send them a warning if some of them begin to deviate from their group.

The rest of the chapter is organized as follows: Section 4.2 surveys the related work. Section 4.3 presents the JiST operations. Section 4.4 and 4.5 describe the JiST algorithms. The application of this framework on trajectory privacy preservation is presented in Section 4.6. Experimental results are presented in Section 4.7, and Section 4.8 contains concluding remarks.

4.2 Related Work

Related previous research work can be largely classified into two areas, Spatio-Temporal Joins, and Location Privacy.

4.2.1 Spatio-Temporal Joins

There is a large body of research on both spatial join [16, 18, 20, 24, 27–29, 46, 48, 49, 80, 95, 109, 112, 116, 117] and temporal join [32, 41, 63, 91, 94, 96, 99, 115, 118] operations, respectively. However, there is very limited work on spatio-temporal Join operations that comprehensively cover both the spatial and temporal aspects of the problem.

Iwerks et al. [50] proposed an algorithm for maintaining a dynamic view of the "spatial semijoin" results as time progresses. The "join" operation, however, focuses on the *present time*, which is essentially a time point.

Jeong et al. [55] experimentally evaluated the performance of several previously proposed join strategies in a spatio-temporal setting. However, the spatio-temporal join operations addressed in this work considers spatial relationships such as *intersects*, *contains* between stationary objects that evolve over time, which is an orthogonal problem to the moving object trajectory join operations we study in this chapter.

Bakalov et al. [8] address the *Window Time-Parameterized Distance Join* problem using symbolic representation [66]. The query operation studied in [8] returns the pairs of objects from two spatio-temporal datasets whose distance between each other remains below a certain threshold value ϵ throughout a time window Δt. This is a very specific instantiation of the broad class of query operations we will cover with the JiST framework, including the Window Trajectory Distance Join operation which cannot be answered using the techniques proposed in [8]. In addition, the techniques proposed in [8] are applicable specifically to historical data, on which a static symbolic representation can be built.

Sun et al. [100] present detailed discussion on selectivity estimation of spatio-temporal join operations, but only focus on a specific operation, namely *predictive time-stamp*

distance join.

The index structure we use in this chapter is a simple extension of the STRIPES structure presented in Chapter 2, which employs the dual transformation techniques [6,59]. The basic idea is to transform a linear trajectory defined by the equation: $\overline{P} = \overline{P}_{ref} + \overline{V}(t - t_{ref})$ in $(D + 1)$-dimensional space (t being the additional dimension) into a point $(\overline{V}, \overline{P}_{ref})$ in $2D$-dimensional dual space. Here, $\overline{V} = (V_1, V_2, \cdots, V_D)$, and $\overline{P}_{ref} = (P_{ref1}, P_{ref2}, \cdots, P_{ref\mathtt{D}})$ are the transformed velocity and reference position vectors.

Since time is monotonically increasing, the value of $P_{ref\mathtt{i}}$ is not bounded, which makes it impossible to build an index that extends into the infinite future. To address this problem, previous works have employed a two-index strategy [54, 59, 71, 83, 105]. This strategy keeps two temporally consecutive index structures in the system, both with lifetime L. For example, if the first index structure is effective within time interval $[0, L)$, then the second index structure will be effective in interval $[L, 2L)$. Objects are required to issue an update every L time units to maintain a valid entry in the index.

4.2.2 Location Privacy

Previous work on location privacy preservation methods have been classified into three categories [67]: user-defined or system-provided policy specification [9, 13, 40], location anonymization [9, 26, 34, 37, 39, 73], and pseudonymity of user identities [13].

Existing methods for preserving location k-Anonymity focus on processing current locations, and are not applicable to trajectories. We define the concept of *Trajectory k-Anonymity*, introduce a set of trajectory privacy policies, and adapt the JiST join operations to provide a new class of *Trajectory Cloaking* algorithms.

4.3 Trajectory Join Operations

In this section we formally define the set of JiST operations. In addition, we also relate these general definitions and previously defined spatio-temporal query operations. As a

Table 4.1: Table of Frequently Used Notations

Notation	Description
D	Dimensionality of data space
$\mathbf{R}$	Query trajectory dataset
$\mathbf{S}$	Target trajectory dataset
r	Trajectory object in dataset $\mathbf{R}$
s	Trajectory object in dataset $\mathbf{S}$
$\overline{r(t)}$	D-dimensional position vector of trajectory object r at time t
$\overline{s(t)}$	D-dimensional position vector of trajectory object s at time t
$\overline{V_r(t)}$	D-dimensional velocity vector of trajectory object r at time t
$\overline{V_s(t)}$	D-dimensional velocity vector of trajectory object s at time t
w	A half-open time interval, i.e., $[t_1, t_2)$
$r(w)$	Segment of trajectory r within interval w
$s(w)$	Segment of trajectory s within interval w
$\mathbf{DIST}(\overline{r(t)}, \overline{s(t)})$	The Euclidean distance between trajectories r and s at time t
I_R	JiST index on dataset $\mathbf{R}$
I_S	JiST index on dataset $\mathbf{S}$
$I_R(t)$	Base index in I_R with $t_{ref} = t$
$I_S(t)$	Base index in I_S with $t_{ref} = t$
L	Lifetime of dual transformed spatial indexes
$I_R(w)$	Base index in I_R with $[t_{ref}, t_{ref} + L) \cap w \neq \emptyset$
$I_S(w)$	Base index in I_S with $[t_{ref}, t_{ref} + L) \cap w \neq \emptyset$
N_R	A node entry in index I_R
N_S	A node entry in index I_S

consequence, we also show that the JiST operations are more powerful and general than previous methods.

To facilitate the discussion, we use the notations described in Table 4.1. Also, for ease of presentation, in the rest of our discussion, we use the terms “objects” and “trajectories” interchangeably, since an object is uniquely associated with a trajectory and vice versa.

4.3.1 Trajectory Distance Join (TDJ)

***Definition* 4.1** (General Trajectory Distance Join (G-TDJ))**.**

Given two point trajectory datasets, the query dataset $\mathbf{R}$ *and the target dataset* $\mathbf{S}$*, and a positive real number* Δd*, the G-TDJ operation finds, for each trajectory* $r \in \mathbf{R}$*, all trajectories* $s \in \mathbf{S}$ *such that* s *is within distance* Δd *of* r *for some time intervals. The formal*

definition is:

$$\mathbf{R} \bowtie_{\Delta d} \mathbf{S} \equiv \{(r, \{(s, \{w\})\}) \quad | \quad r \in \mathbf{R} \wedge s \in \mathbf{S} \wedge$$
$$\forall_{t \in w} DIST(\overline{r(t)}, \overline{s(t)}) \leq \Delta d \wedge$$
$$\neg \exists_{w' \supset w} \forall_{t' \in w'} DIST(\overline{r(t')}, \overline{s(t')}) \leq \Delta d\}$$

Definition 4.1 gives the general case definition of the TDJ operation, which spans the entire time horizon of both datasets **R** and **S**. However, often we are only concerned with a short time window, such as "*five minutes from now*", or "*yesterday between two 2 and 3 PM*". To address these types of questions, we impose a temporal restriction on G-TDJ and introduce the Window Trajectory Distance Join (W-TDJ) operation in Definition 4.2.

***Definition* 4.2** (Window Trajectory Distance Join (W-TDJ))**.**

Given two point trajectory datasets, the query dataset **R** *and the target dataset* **S***, a positive real number* Δd*, and a time window* w*, the W-TDJ*$_s$ *operation finds, for each trajectory* $r \in \mathbf{R}$*, all trajectories* $s \in \mathbf{S}$ *such that each* s *is within distance* Δd *of* r *for some duration within the time window* w*. It is formally defined below.*

$$\mathbf{R} \bowtie_{\Delta d, w} \mathbf{S} \equiv \{(r, \{(s, \{w'\})\}) \quad | \quad r \in \mathbf{R} \wedge s \in \mathbf{S} \wedge$$
$$w' \subseteq w, \forall_{t \in w} DIST(\overline{r(t)}, \overline{s(t)}) \leq \Delta d \wedge$$
$$\neg \exists_{w'' \supset w'} \forall_{t' \in w''} DIST(\overline{r(t')}, \overline{s(t')}) \leq \Delta d\}$$

In relational algebra, the relation between W-TDJ$_s$ and G-TDJ is as follows:

$$\mathbf{R} \bowtie_{\Delta d, w} \mathbf{S} = \pi_{(r,s,w' \cap w)} \left(\sigma_{w' \cap w \neq \phi} (\mu_{\{w'\}} (\mu_{\{s\}} (\mathbf{R} \bowtie_{\Delta d} \mathbf{S}))) \right)$$

In practice, the W-TDJ operation defined in Definition 4.2 is considered to be the more commonly used operation as opposed to the G-TDJ operation, thus we provide detailed

discussion and algorithms for W-TDJ in the rest of this chapter. For simplicity, in the rest of our discussion we use the term TDJ to refer to the W-TDJ operation.

4.3.2 Trajectory kNN Join (TkNNJ)

***Definition* 4.3** (General Trajectory kNN Join (G-TkNNJ))**.**

Given two point trajectory datasets, the query dataset **R** *and the target dataset* **S***, and a positive integer k, the G-TkNNJ operation finds, for all trajectories $r \in \mathbf{R}$, the sets of their k Nearest Neighbors in* **S***, and the time intervals in which the results remain valid. Formally, this operation is defined as:*

$$\begin{aligned}\mathbf{R} \bowtie_k \mathbf{S} \quad \equiv \quad & \{(r, \{w, \{s\}\}) | r \in \mathbf{R} \wedge \{s\} \subseteq \mathbf{S} \wedge \cup_w = W \wedge |\{s\}| = k \wedge \\ & \forall_{w_i, w_j \in \{w\}} w_i \cap w_j = \phi \wedge \\ & \forall_{t \in w} \forall_{s \in \{s\}} \neg \exists_{s' \in \mathbf{S} - \{s\}} DIST(\overline{r(t)}, \overline{s'(t)}) < DIST(\overline{r(t)}, \overline{s(t)}\}\end{aligned}$$

Definition 4.3 gives the general case definition of the TkNNJ operation. Similar to the TDJ operation, the Window Trajectory kNN Join (W-TkNNJ) operation is more commonly used and is formally presented in Definition 4.4. We refer to W-TkNNJ as TkNNJ in the rest of this chapter.

***Definition* 4.4** (Window Trajectory kNN Join (W-TkNNJ))**.**

Given two point trajectory datasets, the query dataset **R** *and the target dataset* **S***, a positive integer k, and a time window w, the W-TkNNJ operation finds, for all $r \in \mathbf{R}$, the sets of their k Nearest Neighbors in* **S***, and the time intervals in which the results remain valid, throughout the time window w. The formal definition is given below.*

$$\begin{aligned}\mathbf{R} \bowtie_{k,w} \mathbf{S} \quad \equiv \quad & \{(r, \{w', \{s\}\}) | r \in \mathbf{R} \wedge \{s\} \subseteq \mathbf{S} \wedge \cup_{w'} = w \wedge \\ & |\{s\}| = k \wedge w' \subseteq w \wedge \\ & \forall_{w_i, w_j \in \{w\}} w_i \cap w_j = \phi \wedge \\ & \forall_{t \in w'} \forall_{s \in \{s\}} \neg \exists_{s' \in \mathbf{S} - \{s\}} DIST(\overline{r(t)}, \overline{s'(t)}) < DIST(\overline{r(t)}, \overline{s(t)}\}\end{aligned}$$

In relational algebra, the relation between W-TkNNJ and G-TkNNJ is as follows:

$$\mathbf{R} \bowtie_{k,w} \mathbf{S} = \pi_{(r,w' \cap w,s)}(\sigma_{w' \cap w \neq \phi}(\mu_{\{w'\}}(\mu_{\{s\}}(\mathbf{R} \bowtie_{k} \mathbf{S}))))$$

4.3.3 Relaxations and Restrictions

There are two aspects to the relaxations and restrictions of the general JiST join operations: the temporal domain and the cardinalities of the datasets in the join operations. We provide brief discussions for each of these aspects below.

4.3.3.1 Temporal Domain

In the temporal domain, there are two factors to consider: the extent of the query window, i.e., the query window size $|w|$; and the placement of the query window, e.g, in the past or sometime in the future.

In the case where $|w| = 0$, the JiST join operations become pure spatial join operations on top of a snapshot of the spatio-temporal database. These operations include Spatial Distance Join operations [20, 46, 49, 95] and Spatial k Nearest Neighbors Join operations [18, 27, 112, 116]. If an additional cardinality restriction k is imposed on the result set of the join operations then the query operation is reduced to the Top-k Spatial Join problem [117].

Depending on whether the query window refers to some time in the past or the future, the JiST join operations evaluate historical or predictive queries.

4.3.3.2 Data Cardinality

Consider the JiST join operations with $|R| = 1$, then these operations are reduced to either time-parameterized range queries or kNN queries for single query points that are applicable both in historical and predictive settings.

4.4 JiST Join Algorithms

In this section we present the JiST join algorithms. We only discuss in detail the JiST Window Trajectory Distance Join, which we call TDJ, and the JiST Window Trajectory *k*NN Join algorithm, which is termed TkNNJ. The extension to the general join operations defined in Sections 4.3.1 and 4.3.2 is straightforward and can be inferred by extending the query window.

Before delving into the algorithms, we make a few assumptions about the data model used in JiST to represent trajectories, and introduce the Time-Parameterized Distance measure between trajectories.

4.4.1 Representing Trajectories in JiST

We make the following assumptions about object movement patterns in JiST.

Assumption 1. A moving object r updates its motion parameters $\overline{r(t)}$ and $\overline{V_r(t)}$ either periodically or when the velocity vector change exceeds a certain threshold since last update, together with the timestamp t. The update information $(\overline{V_r(t)}, \overline{r(t)}, t)$ is stored as a tuple in a table or as an entry in an index.

Assumption 2. In between updates, objects move in a straight line with the same velocity as reported in the most recent update.

Based on Assumption 1, a trajectory r in JiST is represented as the time-ordered sequence $\{(\overline{V_r(t_1)}, \overline{r(t_1)}, t_1), (\overline{V_r(t_2)}, \overline{r(t_2)}, t_2), \cdots\}$.

As a result of Assumption 2, let $w^{\vdash}$ and $w^{\dashv}$ denote the lower and upper boundaries of the time interval w between updates, then a trajectory segment $r(w)$ that starts at time $w^{\vdash}$ and is updated at time $w^{\dashv}$ is represented in JiST as $r(w) \equiv \left(\overline{V_r(w)}, \overline{r(w^{\vdash})}, w\right)$.

Next we present the Time-Parameterized Distance measure between trajectories.

4.4.2 Time-Parameterized Distance Between Trajectories

***Definition* 4.5.** *Using vector operations, the Time-Parameterized Distance (TPD) between trajectories r and s at time t is defined as:*

$$\begin{aligned} TPD(r,s,t) &= DIST\left(\overline{r(t)}, \overline{s(t)}\right) \\ &= \sqrt{\left(\overline{r(t)} - \overline{s(t)}\right)^T \cdot \left(\overline{r(t)} - \overline{s(t)}\right)} \end{aligned} \tag{4.1}$$

We observe that TPD is not defined for two trajectories that do not have any temporal overlap.

Next we provide measures to bound the TPD of trajectories.

Given time t, let $r(w)$ be the segment on trajectory r such that $t \in w$, and let $s(w')$ be the segment on trajectory s such that $t \in w'$. We obtain the following representations of $\overline{r(t)}$ and $\overline{s(t)}$ according to assumption 2.

$$\begin{cases} \overline{r(t)} \equiv \overline{r(w^{\vdash})} + \overline{V_r(w)}t \\ \overline{s(t)} \equiv \overline{s(w'^{\vdash})} + \overline{V_s(w')}t \end{cases} \tag{4.2}$$

Substituting 4.2 into 4.1 and re-organizing yields Equation 4.3,

$$TPD(r,s,t) = \sqrt{\alpha t^2 + \beta t + \gamma} \tag{4.3}$$

where

$$\begin{aligned} \alpha &= \sum_{d=1}^{D}\left(\overline{V_r(w)}_d - \overline{V_s(w')}_d\right)^2 t^2 \\ \beta &= 2\sum_{d=1}^{D}\left(\overline{V_r(w)}_d - \overline{V_s(w')}_d\right)\left(\left(\overline{r(w^{\vdash})}_d - \overline{V_r(w)}_d w^{\vdash}\right) - \left(\overline{s(w'^{\vdash})}_d - \overline{V_s(w')}_d w'^{\vdash}\right)\right) \\ \gamma &= \sum_{d=1}^{D}\left(\left(\overline{r(w^{\vdash})}_d - \overline{V_r(w)}_d w^{\vdash}\right) - \left(\overline{s(w'^{\vdash})}_d - \overline{V_s(w')}_d w'^{\vdash}\right)\right)^2 \end{aligned}$$

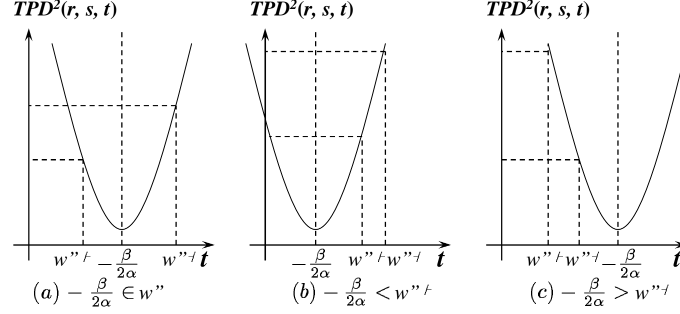


Figure 4.2: Finding $TPD^{\vdash}(r, s, w'')$ and $TPD^{\dashv}(r, s, w'')$

and the subscript d indicates the dimension.

To solve for the minimum and maximum of $TPD^2(r, s, t)$ three cases need to be considered, which can also be observed in Figure 4.2:

(a)(Figure 4.2(a))	$-\frac{\beta}{2\alpha} \in w''$	$TPD^{\vdash}(r, s, w'') = TPD\left(r, s, -\frac{\beta}{2\alpha}\right)$, $TPD^{\dashv}(r, s, w'') = \max(TPD(r, s, w''^{\vdash}), TPD(r, s, w''^{\dashv}))$
(b)(Figure 4.2(b))	$-\frac{\beta}{2\alpha} < w''^{\vdash}$	$TPD^{\vdash}(r, s, w'') = TPD(r, s, w''^{\vdash})$, $TPD^{\dashv}(r, s, w'') = TPD(r, s, w''^{\dashv})$
(c)(Figure 4.2(c))	$-\frac{\beta}{2\alpha} > w''^{\dashv}$	$TPD^{\vdash}(r, s, w'') = TPD(r, s, w''^{\dashv})$, $TPD^{\dashv}(r, s, w'') = TPD(r, s, w''^{\vdash})$

On the other hand, given a distance upper bound Δd, the set of time intervals $\{w\}$ in which two trajectories r and s are no farther than Δd from each other can be obtained by solving ranges for t in the equation: $TPD(r, s, t) \leq \Delta d$.

4.4.3 Naïve Algorithms

In cases where there are no indices available on either **R** or **S**, we provide the Block Nested Loops Join algorithms that employ sequential table scan and sorting techniques.

Block Nested Loops Distance Join (BNL_DJ)

The Block Nested Loops Distance Join algorithm presented in Algorithm 4.1 is fairly straightforward. The algorithm proceeds in two stages: a scan and filter stage (lines 1 and 2), and a join stage.

The algorithm starts with a sequential scan on both tables and uses the query window

Algorithm 4.1: $BNL_DJ(\mathbf{R}, \mathbf{S}, W, \Delta d)$

```
1 R_W ⇐ TableScan(R, W), S_W ⇐ TableScan(S, W);
2 Materialize R_W, S_W ;
3 foreach r_W ⇐ TableScan(R_W) do
4 |   s_W ⇐ TableScan(S_W);
5 |   foreach e_r ∈ r_W do
6 |   |   foreach e_s ∈ s_W do
7 |   |   |   w ⇐ find_overlap_w(e_r.w, e_s.w, W);
8 |   |   |   {w'} ⇐ solve TPD(e_r, e_s, t) ≤ Δd for t;
9 |   |   |   Return result (e_r, (e_s, {w ∩ {w'}}));
```

W as the filtering predicate. Qualifying trajectories are retrieved from the tables and materialized as intermediate views R_W and S_W. During the join operation, trajectories are scanned from R_W one at a time (line 4). For each trajectory r_W retrieved from R_W, another sequential scan is performed on S_W to retrieve trajectories s_W one at a time (line 5). The trajectories r_W and s_W are then broken into segments e_r and e_s with overlapping time intervals, and line 8 directly applies the solution of inequality discussed in Section 4.4.2 to obtain the results.

On the other hand, the Block Nested Loops kNN Join algorithm (BNL_kJ), as is shown in Algorithm 4.2, is slightly more involved, and we will discuss it in more details below.

Block Nested Loops kNN Join (BNL_kJ)

Algorithm 4.2 presents the top level BNL_kJ algorithm. The scan and filter stage on lines 1 and 2 is exactly the same as that in BNL_DJ. However, the join stage of BNL_kJ introduces two new data structures, the Temporal Priority Queue (TPQ) and the Window Priority Queue (WPQ).

Each TPQ is "owned" by a trajectory segment e_r in table **R**. The TPQ structure consists of the following fields: 1) an *owner*, e_r, which is a trajectory segment e_r in table **R** 2) a priority queue that contains instances of the WPQ structure.

The WPQ structure consists of the following fields: 1) a time interval w, of which the lower bound serves as the key for ordering in the TPQ; 2) a priority queue; 3) a *max_tpd*

Algorithm 4.2: $BNL_kJ(\mathbf{R}, \mathbf{S}, W, k)$

```
1  R_W ⇐ TableScan(R, W), S_W ⇐ TableScan(S, W);
2  Materialize R_W, S_W ;
3  foreach r_W ⇐ TableScan(R_W) do
4      s_W ⇐ TableScan(S_W);
5      foreach e_r ∈ r_W do
6          TPQ_r ⇐ new TPQ(e_r);
7          foreach e_s ∈ s_W do
8              w ⇐ find_overlap_w(e_r.w, e_s.w, W);
9              updateTPQ(TPQ_r, w, e_s, k);
10         while wpq ⇐ TPQ_in.DEQUEUE() do refineResults(k, e_r, wpq);
```

Algorithm 4.3: $updateTPQ(TPQ_{in}, w, e_s, k)$

```
1  e_r ⇐ TPQ_in.owner;
2  if notExist(wpq ∈ TPQ_in such that Overlaps(wpq.w, w)) then
3      wpq′ ⇐ new WPQ(w);
4      min_tpd ⇐ TPD⊢(e_r, e_s, w), max_tpd ⇐ TPD⊣(e_r, e_s, w);
5      wpq′.ENQUEUE(min_tpd, max_tpd, e_s);
6      TPQ_in.ENQUEUE(wpq′);
7  else foreach wpq ∈ TPQ_in AND Overlaps(wpq.w, w) do
8      w′[3] ⇐ {wpq.w − w, wpq.w ∩ w, w − wpq.w};
9      for i ⇐ 1 to 3 do
10         wpq′[i] ⇐ new WPQ(w′[i]);
11     for i ⇐ 2 to 3 do
12         min_tpd ⇐ TPD⊢(e_r, e_s, w′[i]);
13         max_tpd ⇐ TPD⊣(e_r, e_s, w′[i]);
14         wpq′[i].ENQUEUE(min_tpd, max_tpd, e_s);
15     while e′_s ⇐ wpq.DEQUEUE() do
16         for i ⇐ 1 to 2 do
17             min_tpd ⇐ TPD⊢(e_r, e′_s, w′[i]);
18             max_tpd ⇐ TPD⊣(e_r, e′_s, w′[i]);
19             if sizeof(wpq′[i]) < k OR min_tpd < wpq′[i].max_tpd then
                   wpq′[i].ENQUEUE(min_tpd, max_tpd, e′_s);
20     for i ⇐ 1 to 3 do TPQ_in.ENQUEUE(wpq′[i]);
```

field which serves as the pruning threshold.

Entries in the priority queue of a WPQ contain the following information: 1) trajectory segment e_s from table $\mathbf{S}$; 2) $min_tpd = TPD^{\vdash}(e_r, e_s, w)$; and 3) $max_tpd = TPD^{\dashv}(e_r, e_s, w)$. The min_tpd field is used as the sort key in the priority queue of WPQ, and the max_tpd

Algorithm 4.4: $refineResults(k, e_r, wpq)$

```
1  if wpq follows Total Ordering then  Return result (e_r, wpq);
2  else
3      TQ ⇐ new PriorityQueue();
4      foreach e_s ∈ wpq do
5          foreach e'_s ∈ wpq AND e'_s > e_s do
6              if e'_s.min_tpd > e_s.max_tpd then break;
7              else  t ⇐ solve Equation TPD(e_s, e'_s, t) = 0 for t;
8              if t ∈ wpq.w then  TQ.ENQUEUE(t);
9      t⊢ ⇐ wpq.w⊢;
10     while t ⇐ TQ.DEQUEUE() do
11         wpq' ⇐ new WPQ([t⊢, t));
12         foreach e_s ∈ wpq do
13             min_tpd ⇐ TPD⊢(e_r, e_s, [t⊢, t));
14             max_tpd ⇐ TPD⊣(e_r, e_s, [t⊢, t));
15             if sizeof(wpq') < k OR min_tpd < wpq'.max_tpd then
16                 wpq'.ENQUEUE(min_tpd, max_tpd, e_s);
17             refineResults(k, e_r, wpq');
18         t⊢ ⇐ t ;
```

field is used by the WPQ to set the pruning threshold for result retrieval.

The *updateTPQ* procedure is presented in Algorithm 4.3. Intuitively, *updateTPQ* proceeds in two stages: in the first stage, overlapping time intervals are identified between the interval (w) of the incoming entry e_s and those (w') of the the existing WPQs in a TPQ, and new time intervals ($w' - w$, $w' \cap w$, and $w - w'$) together with corresponding WPQs are generated; in the second stage, entries in the old WPQ are re-distributed among the new WPQs with an overlapping time interval. During the re-distribution stage, the *min_tpd* and *max_tpd* fields of the entries are updated with respect to the new time intervals, and un-qualified entries are pruned.

The *refineResults* function shown in Algorithm 4.4 processes a single WPQ, *wpq*. It essentially identifies all the intersection points among all segments e_s, splits the time interval w of *wpq* into even smaller time intervals within which none of the e_s segments intersect with each other, and thus follow a *Total Ordering*. Finally new WPQs are constructed for the smaller time intervals, entries in *wpq* are re-distributed into the new

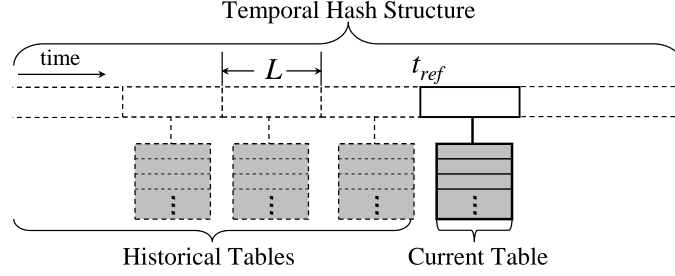


Figure 4.3: The JiST Time Partitioned Storage Model

WPQs and pruned at the same time. WPQs that follow the Total Ordering are then reported together with the TPQ *owner* e_r as results.

We note that based on the linear movement assumption in Section 4.4.1, the proposed BNL_DJ and BNL_kJ algorithms are applicable to both historical and predictive settings, which only differ in the algorithms in the range of query window W to be examined.

4.4.3.1 Time Partitioned Block Nested Loops Join

The main drawback of the BNL algorithms is the sequential scanning and sorting of large tables, which are both CPU and I/O inefficient. However, we observe that the most common JiST join queries are window queries that only retrieve results within a time window specified by the query. In this section we slightly modify the storage model in JiST and introduce the more efficient Time Partitioned Block Nested Loops (TPBNL) join algorithms.

To facilitate the TPBNL join algorithms, we introduce an in-memory hash structure that splits time into intervals of length L, each corresponding to a table that stores trajectories with time stamp falling within the interval. Figure 4.3 depicts the new data storage model.

Using the new Time-Partitioned storage model, TPBNL algorithms are straightforward extensions of the BNL algorithms.

The TPBNL_DJ algorithm presented in Algorithm 4.5 simply identifies data tables R_w and S_w with time intervals overlapping query window W and executes the BNL_DJ

Algorithm 4.5: $TPBNLDJ(\mathbf{R}, \mathbf{S}, W, \Delta d)$

1 $\{w\} \Leftarrow find_overlap_w(\mathbf{R}, \mathbf{S}, W)$;
2 **foreach** $w \in \{w\}$ **do** $BNL_DJ(R_w, S_w)$;

Algorithm 4.6: $TPBNL_kJ(\mathbf{R}, \mathbf{S}, W, k)$

1 $\{w\} \Leftarrow find_overlap_w(\mathbf{R}, \mathbf{S}, W)$;
2 **foreach** $w \in \{w\}$ **do** $BNL_kJ(R_w, S_w)$;

algorithm on these tables.

Similar to TPBNL_DJ, the TPBNL_kJ algorithm in Algorithm 4.6 is a simple extension of the BNL_kJ algorithm, which executes the BNL_kJ algorithm on partition tables of **R** and **S** with overlapping time intervals.

4.4.4 Dual Index Based Algorithms

The naïve algorithms presented in Section 4.4.3 are based on sequential table scan techniques and do not make any assumptions about index structures. Often this is not the case, since various efficient index structures have been proposed in the past to speed up query processing on trajectory data ([54, 65, 83, 104, 105]). In this section, we take advantage of existing indexing techniques and propose the far more efficient JiST Dual Index Based join algorithms.

Although the algorithms we present in this section do not rely on any specific index structure, they do require that it provide certain features, which we discuss in the following subsection.

4.4.4.1 Index Requirements in JiST

In order for the proposed algorithms in this section to work, we impose the following requirements on the underlying index structure: (1) The underlying index uses a two-level indexing scheme, with an in-memory temporal hash structure on the top level, that splits time into intervals of length L, and a tree structure on the bottom level corresponding to each time interval in the hash structure; (2) The bottom level index trees, termed *base*

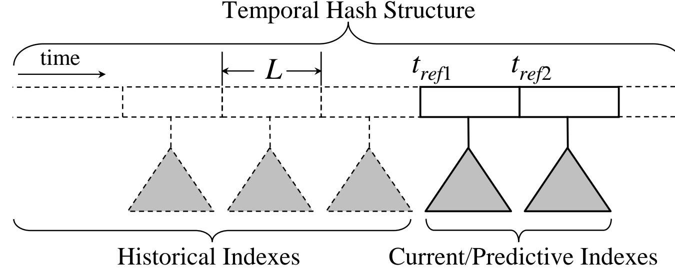


Figure 4.4: The General JiST Index Structure

indexes, employ the dual transform technique [6, 59] to represent trajectories; (3) The underlying index supports the concept of Dual-transformed Minimum Bounding Rectangle (D-MBR) with Validity Interval (VI), which we define in the next section. Figure 4.4 depicts the general index structure assumed by the JiST framework.

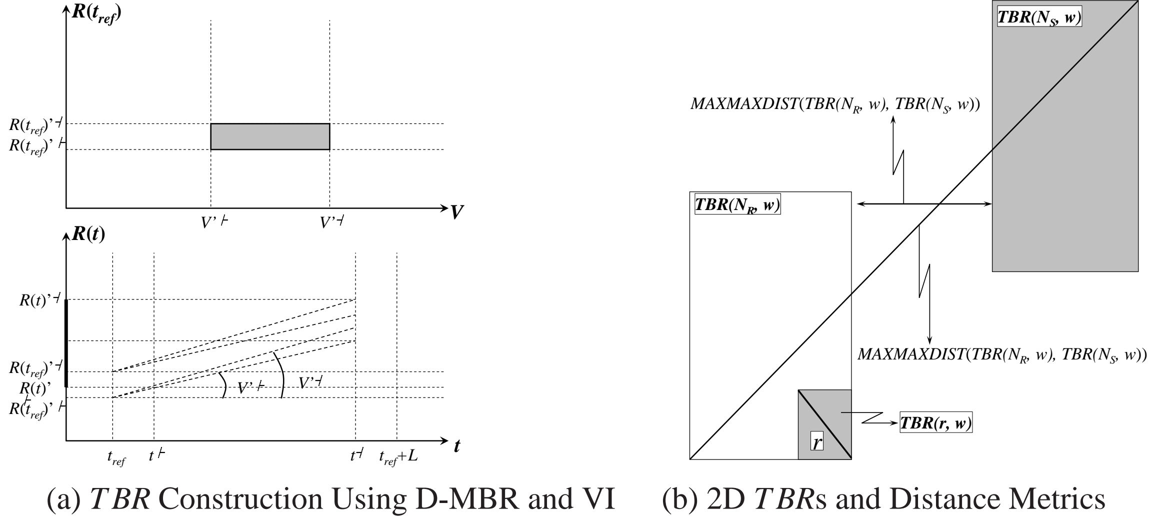
The reasons for choosing a dual transform index structure are two-fold: 1) dual transform technique enables the integration of predictive and historical index structures into one general indexing framework, and 2) dual transform technique enables processing of both historical and predictive queries in a similar fashion.

4.4.4.2 Time-parameterized Bounding Regions and Distance Metrics

In this section we first define the concept of Dual-transformed Minimum Bounding Rectangle (D-MBR) with Validity Interval (VI) within the context of dual transformation, then we proceed to introduce the notion of Time-parameterized Bounding Region and the relevant distance metrics that will be used as pruning criteria in the index based join algorithms.

Using dual transform technique, a linear trajectory segment $r(w)$ can be represented as a point in dual transform space coupled with the time interval w of the segment, namely, $\left(\overline{V_r}, \overline{r(t_{ref})}, w\right)$, where w is called the *Validity Interval (VI)* of this trajectory segment.

***Definition* 4.6.** *Let* $\{r(w)\} \equiv \left\{\left(\overline{V_r}, \overline{r(t_{ref})}, w\right)\right\}$ *represent a set of dual transformed trajectory segments that share the same reference time* t_{ref}*. The Dual-transformed Minimum Bounding Rectangle (D-MBR) of* $\{r(w)\}$ *is defined as* $M\left(\overline{V^{\vdash}}, \overline{V^{\dashv}}, \overline{R(t_{ref})}^{\vdash}, \overline{R(t_{ref})}^{\dashv}\right)$*, where*



(a) TBR Construction Using D-MBR and VI (b) 2D TBRs and Distance Metrics

Figure 4.5: TBR Construction and Distance Metrics

$\overline{V}^{\vdash} = \min\{\overline{V_r}\}$, $\overline{V}^{\dashv} = \max\{\overline{V_r}\}$, $\overline{R(t_{ref})}^{\vdash} = \min\{\overline{r(t_{ref})}\}$, $\overline{R(t_{ref})}^{\dashv} = \max\{\overline{r(t_{ref})}\}$. *And the VI of M is defined as* $VI = \bigcup\{w\}$.

With the D-MBR and VI information stored in the index structure, we are able to infer the Time-parameterized Bounding Region (TBR) for an index entry. We start with the internal index nodes.

Let $M_R\left(\overline{V}^{\vdash}, \overline{V}^{\dashv}, \overline{R(t_{ref})}^{\vdash}, \overline{R(t_{ref})}^{\dashv}\right)$ denote the D-MBR of an internal node N_R in dual transformed index $I_R(t_{ref})$, and let W denote the VI of M_R, the Time-parameterized Bounding Region $TBR(N_R, W)$ can be inferred in the following set of equations:

$$\begin{cases} \overline{R(t)}^{\vdash} & = & \overline{R(t_{ref})}^{\vdash} + \overline{V}^{\vdash}(W^{\vdash} - t_{ref}) \\ \overline{R(t)}^{\dashv} & = & \overline{R(t_{ref})}^{\dashv} + \overline{V}^{\dashv}(W^{\dashv} - t_{ref}) \end{cases} \tag{4.4}$$

In Equation Set 4.4 the D dimensional vector $\overline{R(t)}^{\vdash}$ denotes the lower bounds of $TBR(N_R, W)$ for all $t \in W$. Similarly, the vector $\overline{R(t)}^{\dashv}$ denotes the upper bounds.

The geometric intuition of $TBR(N_R, W)$ is that it covers the spatial region that all objects enclosed in N_R may traverse during time interval W. Figure 4.5(a) illustrates the inference of a one-dimensional TBR. As is shown, the top portion of the figure indicates the bounding regions in dual-space, while the bottom portion of the figure shows the

inferred TBR regions as a dark thick line segments on the $R(t)$ axis.

Another observation that we make from Figure 4.5(a) is that since the one-dimensional TBR is a continuous spatial interval, TBRs in multi-dimensional space take the shape of hyper-rectangle regions. Traditional MBR distance metrics such as MINMINDIST and MAXMAXDIST are also applicable on TBRs.

***Definition* 4.7.** *Let $TBR(N_R, W)$ and $TBR(N_S, W)$ be two inferred time-parameterized bounding regions across a time interval W for index nodes N_R and N_S, then the metric MINMINDIST($TBR(N_R, W)$, $TBR(N_S, W)$) is defined as the minimum distance between any point within $TBR(N_R, W)$ and any point within $TBR(N_S, W)$ over time interval W.*

The MINMINDIST metric between TBRs gives the lower bound of the distance between any object within N_R and any object within N_S, during time interval W.

***Definition* 4.8.** *Let $TBR(N_R, W)$ and $TBR(N_S, W)$ be two inferred time-parametereized bounding regions across a time interval W for index nodes N_R and N_S, then the metric MAXMAXDIST($TBR(N_R, W)$, $TBR(N_S, W)$) is defined as the maximum distance between any point within $TBR(N_R, W)$ and any point within $TBR(N_S, W)$*

The MAXMAXDIST metric between TBRs gives the upper bound of the distance between any object within N_R and any object within N_S, during time interval W

The computation of the MINMINDIST and MAXMAXDIST metrics between TBRs is exactly the same as that between traditional MBRs([28]), which we will omit to avoid repetition.

For a trajectory segment $r(w)$, $TBR(r, w)$ is simply the MBR that bounds the trajectory within time interval w.

Figure 4.5 shows example TBRs of two internal nodes N_R and N_S, and a trajectory r for a time interval w. The MINMINDIST and MAXMAXDIST metrics between $TBR(N_R, w)$ and $TBR(N_S, w)$ are also shown in the figure.

Since TBRs are defined over a time interval W, the definition of TPD at an exact time point t is not well-defined. However, the lower and upper bounds of TPD between TBRs can be determined as follows:

$TPD^{\vdash}(TBR(N_R, W), TBR(N_S, W)) = MINMINDIST(TBR(N_R, W), TBR(N_S, W))$, and

$TPD^{\dashv}(TBR(N_R, W), TBR(N_S, W)) = MAXMAXDIST(TBR(N_R, W), TBR(N_S, W))$.

Next we proceed to present the JiST Dual Index Based Join algorithms, JiST TDJ and JiST TkNNJ.

4.4.4.3 JiST TDJ Algorithm

As is presented in Algorithm 4.7, JiST TDJ between two JiST indexes I_R (the querying index) and I_S (the target index) proceeds in two steps: first the overlapping time intervals $\{w\}$ of the two indexes and the query window W are gathered, then for each $w \in \{w\}$, the underlying spatial indexes $I_R(w)$ and $I_S(w)$ are retrieved and Distance Join algorithms are used to traverse the two indexes and proceed with the join operation based on distance metrics on TBRs introduced in the previous section.

A variety of methods are applicable to the index traversal. However, previous research [46, 49] has concluded that *breadth-first incremental traversal method* proves the most efficient in Distance Join processing, therefore we will use this method in our JiST TDJ algorithm presented in Algorithm 4.7.

During the iterations of the Distance Join operation, a global priority queue Q is used for ordering intermediate join entries. These entries consist of three fields: 1) e_r, an entry from index $I_R(w)$; 2) e_s, an entry from index $I_S(w)$; and 3) *min_tpd*, computed as $TPD^{\vdash}(TBR(e_r, w), TBR(e_s, w))$ The intermediate join entries are ordered by their *min_tpd* field inside Q and are pruned if $min_tpd > \Delta d$ before they are enqueued in Q.

Intermediate join entries in Q are expanded and processed in a *bi-directional* fashion, i.e., if both entries e_r and e_s are internal nodes, they are both expanded and entries within them are processed in pairs recursively. If only one of these entries is an internal node,

Algorithm 4.7: $JiST_TDJ(I_R, I_S, W, \Delta d)$

```
1  {w} ⇐ find_overlap_w(I_R, I_S, W);
2  foreach w ∈ {w} do
3      Q ⇐ new PriorityQueue();
4      Q.ENQUEUE(I_R(w).root, I_S(w).root);
5      while QEntry ⇐ Q.DEQUEUE() do
6          if QEntry.e_r is OBJECT and QEntry.e_s is OBJECT then
7              w' ⇐ find_overlap_w(e_r.VI, e_s.VI, w);
8              {w''} ⇐ solve TPD(e_r, e_s, t) ≤ Δd for t;
9              Return result (e_r, (e_s, {{w''} ∩ w'}));
10         else if QEntry.e_r is OBJECT then
11             foreach e'_s ∈ e_s do
12                 w' ⇐ find_overlap_w(e_r.VI, e'_s.VI, w);
13                 if TPD⊦(TBR(e_r, w'), TBR(e_s', w')) ≤ Δd then
14                     Q.ENQUEUE(e_r, e'_s)
15         else
16             foreach e'_r ∈ e_r, e'_s ∈ e_s do
17                 w' ⇐ find_overlap_w(e'_r.VI, e'_s.VI, w);
18                 if TPD⊦(TBR(e_r', w'), TBR(e_s', w')) ≤ Δd then
19                     Q.ENQUEUE(e'_r, e'_s)
```

then it is expanded and its child entries are paired up with the other entry and processed.

The steps described above are called the filtering stage of the JiST TDJ algorithm.

When both entries e_r and e_s are objects, the inequality discussed in Section 4.4.2 is solved and corresponding time intervals, if any, are reported together with the pair of objects as results. This constitutes the refinement stage.

4.4.4.4 JiST TkNNJ Join Algorithm

In Chapter 3 we have drawn the conclusion that the *depth-first bi-directional* method yields the best performance in evaluating the *AkNN* operation, attributed to its low memory consumption and fast descent down both the querying and target indexes. We recognize that the JiST T*k*NNJ Join problem is essentially a complex time-parameterized *AkNN* problem, so in our algorithms we adopt this method for traversing the join indexes and expanding intermediate entries during the filtering stage.

Algorithm 4.8: $JiST_TkNNJ(I_R, I_S, W, k)$

```
1  {w} ⇐ find_overlap_w(I_R, I_S, W);
2  foreach w ∈ {w} do
3      foreach e_r ∈ I_R(w).root do
4          TPQ_r ⇐ new TPQ(e_r);
5          foreach e_s ∈ I_S(w).root do
6              Find overlapping time interval w′ between ;
7              w′ ⇐ find_overlap_w(e_r.VI, e_s.VI, w);
8              updateTPQ(TPQ_r, w′, e_s, k);
9          TPkNN(TPQ_r, k);
```

Algorithm 4.9: $TPkNN(TPQ_{in}, k)$

```
1  e_r ⇐ TPQ_in.owner;
2  if e_r is OBJECT then
3      while wpq = TPQ_in.DEQUEUE() do
4          if wpq contains all objects then refineResults(k, e_r, wpq);
5          else while e_s = wpq.DEQUEUE() do
6              w ⇐ find_overlap_w(e_r.VI, e_s.VI);
7              updateTPQ(TPQ_in, w, e_s, k);
8  else foreach e′_r ∈ e_r do
9      TPQ_r′ ⇐ new TPQ(e′_r);
10     foreach wpq ∈ TPQ_in do
11         foreach e_s ∈ wpq do
12             if e_s is OBJECT then
13                 w ⇐ find_overlap_w(e′_r.VI, e_s.VI);
14                 updateTPQ(TPQ_r′, w, e_s, k);
15             else foreach e′_s ∈ e_s do
16                 w ⇐ find_overlap_w(e′_r.VI, e′_s.VI);
17                 updateTPQ(TPQ_r′, w, e′_s, k);
18     TPkNN(TPQ_r′, k);
```

Algorithm 4.8 presents the top-level JiST T*k*NNJ algorithm. The T*k*NNJ algorithm also makes use of the data structures TPQ and WPQ introduced in Section 4.4.3 for intermediate filtering and result retrieval. Similar to the JiST TDJ algorithm, the T*k*NNJ algorithm also proceeds in two phases. In the first phase the overlapping time intervals between I_R and I_S are gathered and organized in an ordered set $\{w\}$. After that, the second phase is lauched and the dual transformed indexes $I_R(w)$ and $I_S(w)$ are retrieved

for each $w \in \{w\}$, their root nodes are expanded *bi-directionally*. At this stage, TPQs are constructed for each entry e_r in the root node of $I_R(w)$ and are updated with entries e_s from the root node of $I_S(w)$

We show the $TPkNN$ procedure in Algorithm 4.9. This is the filtering stage of the JiST TkNNJ algorithm. $TPkNN$ recursively traverses the indexes $I_R(w)$ and $I_S(w)$, expands internal nodes from both indexes, constructs new TPQs for each newly expanded entry from $I_R(w)$, and updates these new TPQs with entries expanded from $I_S(w)$. The procedure stops when all priority queue entries of all WPQs are objects from $I_S(w)$ at which point the $refineResults$ function is invoked, which indicates the beginning of the refinement stage of the algorithm.

We note that attributed to the nature of the dual transform technique, *the JiST TDJ and TkNNJ algorithms are applicable to both historical and predictive settings*.

4.5 Using Indices in JiST

There currently exist several choices for the index structure that meet the requirements outlined in Section 4.4.4.1: the BB^x-index [65], STRIPES (Chapter 2), and the TPR*-tree [105]. Although the adaptation to the JiST framework is fairly straightforward for all the structures above, we chose STRIPES as the base index in our implementation, for the following reasons: (1) In Chapter 2 we have shown that STRIPES is more efficient than TPR*-tree in both updates and query support; (2) The BB^x-index is built on top of the B^x-tree [54], which uses space-filling curves. This adds to the difficulty of adapting the dual transform MBR; (3) The STRIPES index combines the dual-transform technique with the multi-dimensional quadtree structure, which naturally lends to the ease of adaptation to the JiST framework.

Since the initial design of the STRIPES index structure was targeted specifically at predictive query processing, it fits perfectly into the predictive setting of the JiST framework. However, in order for STRIPES to process historical trajectory data, index

entries in STRIPES need to be augmented with the VI information.

Furthermore, we observe that the *grid* information in STRIPES does not provide the tight bounds that are required by the JiST join algorithms for efficient filtering, thus it is necessary to add D-MBR information to STRIPES index entries.

In the following we discuss the adaptation of STRIPES to the JiST framework, which consists of two aspects, the addition of the in-memory temporal hash structure, and the augmentation and maintenance of index entries with D-MBR and VI.

Adding the Temporal Hash: In accordance with the index requirements in JiST (Section 4.4.4.1, STRIPES forms the base indexes on the bottom of the two-level JiST index. On the top level, time is split into intervals of length L, each corresponding to the lifetime of the underlying STRIPES structure (Figure 4.4). An in-memory hash structure is maintained for identifying the base indexes associated with the hashed time intervals. Given a specific time instance $t \in [t_{refi}, t_{refi} + L)$, denoting system initialization time as t_0, the hash function $H(t) = \lfloor \frac{(t-t_0)}{L} \rfloor - 1$ is used to identify the underlying STRIPES structure corresponding to time interval $[t_{refi}, t_{refi} + L)$. As is shown in Figure 4.4, the two-index strategy in STRIPES is retained in JiST for current/predictive base indexes.

Augmenting and Maintaining Index Entries: Augmenting index entries in STRIPES is straightforward. For a leaf index entry in STRIPES, which is a point in the dual transformed space, VI is simply the time interval it remains within a leaf node, and D-MBR is reduced to the point itself. On the other hand, the D-MBR and VI of an internal index entry in STRIPES follow their definitions in a straightforward fashion and represent the union of the corresponding measure of all enclosed child entries. Unlike the updates in STRIPES which result in the deletion of old entries and insertion of new entries, an update in JiST is executed as an update of the old entry (including the update of D-MBR and VI) followed by the insertion of the new entry.

4.6 Trajectory Privacy Preservation

In this section we present the usage scenario of the JiST operations for trajectory privacy preservation.

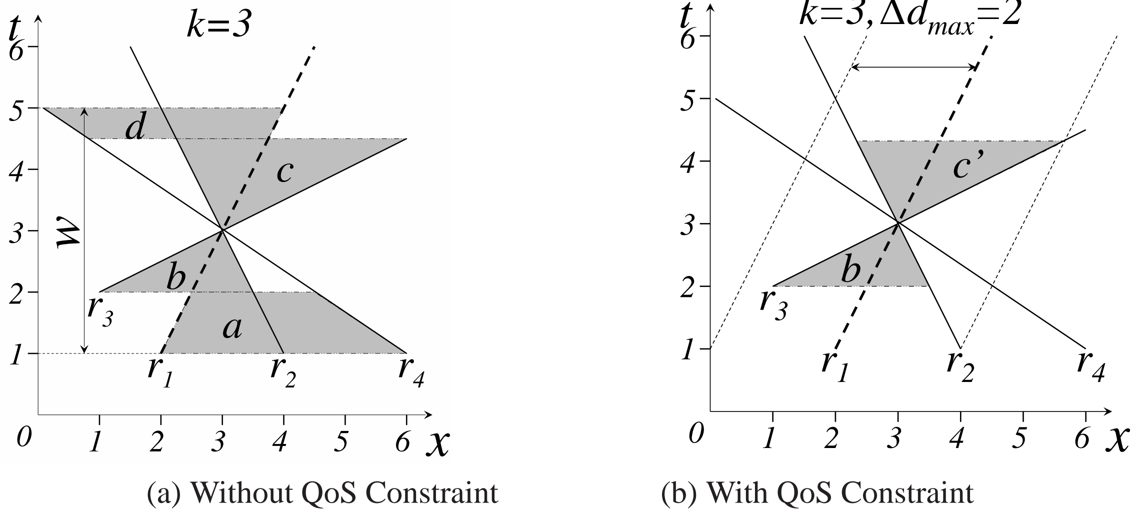
4.6.1 Trajectory k-Anonymity

A location $\overline{r(t)}$ is k-Anonymous if and only if it is indistinguishable from $k-1$ other locations [39]. Location k-Anonymity is also referred to as *Spatial k-Anonymity* [37], so we use these terms interchangeably. The concept of spatial k-Anonymity has been commonly used as a main factor in specifying location privacy policies. However, spatial k-Anonymity is applicable only to snapshots of user locations [37], whereas a trajectory is usually represented as a time-stamped series of locations. Consequently, spatial k-Anonymity can be used as a building block towards the concept of trajectory k-Anonymity, which must take into consideration the effect of constant location change of mobile objects with the evolution of time, as is presented in Definition 4.9.

***Definition* 4.9** (Full Trajectory k-Anonymity)**.**

A trajectory r is k-Anonymous if and only if, at any time point t within the lifetime of r, location $\overline{r(t)}$ *is k-Anonymous.*

For simplicity, Figure 4.6(a) illustrates a one-dimensional example of Full Trajectory k-Anonymity. Four objects $r_1 ... r_4$ are shown in the figure. Objects r_1, r_2, and r_4 start at time $t = 1$, at locations $r_1(1) = 2$, $r_2(1) = 4$, and $r_4(1) = 6$, with velocities $v_{r1}(1) = 0.5$, $v_{r2} = -0.5$, and $v_{r4} = -1.5$, respectively. Object r_3 enters the system at time $t = 2$, at location $r_2(2) = 1$, with velocity $v_{r3} = 2$. The shaded regions a, b, c, and d are derived from the self-TkNNJ result set for object r_1 and compose the k-Anonymity region for trajectory r_1 from time 1 to time 5. Details of deriving trajectory k-Anonymity regions from the TkNNJ result set is discussed in Subsection 4.6.3. Note that no k-Anonymity region is shown for r_1 between time 5 and time 6, as k-Anonymity is undefined for r_1 during this



(a) Without QoS Constraint (b) With QoS Constraint

Figure 4.6: Trajectory k-Anonymity with TkNNJ

time period. As a result, as per the definition of Full Trajectory k-Anonymity , trajectory r_1 in its entirety cannot be considered as k-Anonymous.

In reality, however, the lifetime requirement of Full Trajectory k-Anonymity is often overly stringent and usually impossible to achieve. Furthermore, most of the time LBS applications only require access to partial trajectory data over certain time intervals. It is therefore desirable to define trajectory k-Anonymity over a temporal window, which is presented in Definition 4.10

***Definition* 4.10** (Window Trajectory k-Anonymity)**.**

A trajectory r is k-Anonymous with respect to time window w if and only if, at any time point $t \in w$, location $\overline{r(t)}$ is k-Anonymous.

An example of *Window Trajectory k-Anonymity* is also shown in Figure 4.6(a), where the temporal window w is set between time 1 and time 5, within which the k-Anonymity region for trajectory r_1 is well defined and is shown as the composite shaded region. Thus r_1 is considered k-Anonymous within time window w. We will use the temporal window $w = [1, 5)$ in all the examples for the rest of this section.

The concept of *Window Trajectory k-Anonymity* allows piece-wise disclosure of trajectory data with certain privacy guarantees within temporal windows in which *Window*

Trajectory k-Anonymity is preserved.

4.6.2 Trajectory Privacy Policies

Similar to location privacy, there are two dimensions concerning the specification of trajectory privacy policies, namely the *scope* of the policies and the tradeoff between privacy preservation and quality of service (QoS) of LBS applications. The scope of the privacy policies refers to whether the privacy policies are specified by individual users or are applicable to the entire system. The tradeoff between privacy preservation and LBS QoS is due to users' intrinsic desire to receive as high-resolution information as possible from the LBS without divulging too much of their own information.

Set in the context of trajectory data publication, we focus on system-wide privacy policies and consider three constraints in the specification of trajectory privacy policies: (1) the time window w within which the trajectory data is to be disclosed; (2) the trajectory k-Anonymity constraint k; and (3) the maximum spatial cloaking region constraint Δd_{max}, which dictates that the maximum distance between the target trajectory and the rest of trajectories in the k-Anonymity region must not exceed Δd_{max} at any time point within time window w.

The trajectory k-Anonymity constraint parameter k specifies the level of privacy to be provided by the policy. Generally speaking, larger value of k indicates more uncertainty in pinpointing a specific trajectory, thus providing higher level of privacy. There are exceptions, however, to this general rule of thumb, e.g., when all k objects reside in exactly the same location for a shared period of time w, trajectory k-Anonymity alone will fail to prevent the identification of a specific trajectory within time window w. We provide heuristics in presenting k-Anonymous trajectories in Subsection 4.6.4 to alleviate this problem.

On the other hand, the need to comply with the trajectory k-Anonymity constraint may result in overly large k-Anonymity regions which indicates lower resolution of the

trajectory information provided by the LBS application, and as a result, QoS of the LBS applications will often suffer. In the past, this conflict has usually been addressed by introducing spatial constraints in the form of rectangular regions [9, 73]. However, we argue that in the context of trajectory privacy preservation, a single distance constraint has the following advantages over rectangular regions:

- Simplicity: from the point of view of the LBS providers and users, a single distance constraint is much simpler to represent than multi-dimensional rectangular regions;
- Specificity: the specification of a single distance constraint in itself has multi-dimensional implications, as it is usually a well-defined function of spatial and temporal coordinates, which has been shown in Section 4.4;
- Practicality: intuitively it is more sensible and much simpler for mobile users to dictate their privacy policies in terms of a circular region centered at their current locations, e.g., "I would like to keep my location 10-Anonymous within 1 mile of my current location.", as opposed to having to specify terms such as "1 mile north and south to my current location, and 1 mile east and west to my current location".

In the rest of this section we use the three tuple $(w, k, \Delta d_{max})$ to represent trajectory privacy policy, where w and k are mandatory and together they specify the time window in which Window Trajectory k-Anonymity must hold. The distance constraint Δd_{max}, on the other hand, indicates the QoS guarantee level and is set to $NULL$ when QoS is not a concern.

4.6.3 Trajectory Cloaking with JiST

Trajectory Cloaking refers to the process of determining the time-parameterized spatial region(s) for a trajectory such that the requirements imposed by a set of privacy policies are fulfilled.

As opposed to location cloaking algorithms [9, 26, 34, 35, 37, 39, 57, 73] that are

designed to find the cloaking regions for single locations on a snapshot of user locations at a specific time point, trajectory cloaking algorithms need to take into consideration both spatial and temporal domains and compute the trajectory cloaking regions (TCRs) of trajectories. As we will show in later subsections, the TCRs are derived from sets of trajectory segments that conform to the trajectory privacy policy with respect to a given trajectory r for certain periods of time. These sets of trajectory segments are called the Trajectory Cloaking Set (TCS) with respect to r, and is defined as follows.

***Definition* 4.11** (Trajectory Cloaking Set)**.**

The Trajectory Cloaking Set of a trajectory r within a time window w, expressed as $TCS_r = \{(w', \{r'\}, n)\}$*, is defined as a set of time-ordered tuples each composed of a time window* $w' \subset w$ *and a set of trajectory segments* $\{r'\}$ *of cardinality n satisfying certain trajectory privacy policy specification with respect to r within time window* w'.

Next we show that the TkNNJ and TDJ operations provided by the JiST framework fit naturally in the task of trajectory cloaking and require minimum post-processing efforts.

4.6.3.1 TkNNJ Cloaking

Recall that in Section 4.3, the result set of the TkNNJ for a particular trajectory r is presented as the tuple $(r, \{w', \{s\}\})$, which is essentially the TCS of r. Therefore the adaptation of the result set obtained from the TkNNJ operation to achieve trajectory k-Anonymity for a trajectory r is straightforward.

***Assertion* 4.1.** *For trajectory r, its result set from the operation JiST_TkNNJ*(I_R, I_R, w, k) *automatically satisfies* Window Trajectory k-Anonymity *with constraints specified by privacy policy* $(w, k, NULL)$.

Proof. By the definition of TkNNJ, the result set for r contains, at any time point $t \in w$, a set of k nearest neighbors for object r, including r itself, due to the fact that the JiST_TkNNJ(I_R, I_R, w, k) operation indicates a self-join of the index on trajectory data **R**.

Algorithm 4.10: $Trim_Cloaking_w(r, w, \{r'\}, \Delta d_{max})$

1 $w' \Leftarrow w$;
2 **foreach** $r' \in \{r'\}$ **do**
3     $w'' \Leftarrow$ solve TPD(r, r', t) $\leq \Delta d_{max}$ for range of t;
4     $w' \Leftarrow w' \cap w''$;
5 Return w';

Thus, at any time point $t \in w$, location k-Anonymity holds for $\overline{r(t)}$, which by the definition of *Window Trajectory k-Anonymity*, suggests that *Window Trajectory k-Anonymity* with constraint parameters (w, k) must hold for r. □

Assertion 4.1 implies that if the QoS constraint Δd_{max} is not required, i.e. $\Delta d_{max} = NULL$ in the privacy policy, then the JiST TkNNJ operation can be directly applied on a trajectory data set **R** to produce TCS for all trajectories, from which trajectory k-Anonymity regions can be easily derived. An example of TCRs on one-dimensional trajectory data derived directly from the result set of TkNNJ is shown in Figure 4.6(a). Following Algorithm 4.8, one can obtain the result set for r_1 from operation JiST_TkNNJ(I_R, I_R, $[1, 5)$, 3). As can be observed from Figure 4.6(a), besides r_1 itself, trajectories r_2 and r_4 remain in the result set from time 1 to 2, then r_3 replaces r_4 from time 2 to slightly past time 4, and is then disconnected from the system and replaced by r_4 until time 5. Note that TCRs a, b, c, and d shown in the figure correspond to time intervals in which the result set for r_1 remains un-changed, and the spatial boundaries are simply derived by connecting the MBR of locations at the beginning and the end of the time interval, which can be easily obtained from the JiST_TkNNJ result set.

When the privacy policy is augmented with the additional QoS constraint Δd_{max}, i.e. $\Delta d_{max} \neq NULL$, however, additional filtering method is required to ensure that both k-Anonymity and QoS constraints are satisfied. Since the result set obtained from the JiST_TkNNJ algorithm is by itself sliced into continuous sub-windows $\{w'\}$ of w, within each of which *Window Trajectory k-Anonymity* is ensured for r, the filtering algorithm needs only trim each of the sub-windows w', if necessary, to a smaller sub-window w'', in

Algorithm 4.11: $TDJ_post_process(r, \{(r', \{w'\})\})$

```
1 Trajectory Cloaking Set tcs ⇐ ∅;
2 foreach (r', {w'}) ∈ {(r', {w'})} do
3     foreach w' ∈ {w'} do Scan_Merge_Split((w', r'), tcs);
4 Return tcs;
```

which the TPD between all trajectories r' in the result set (r itself included) and r is no greater than Δd_{max}. We call this procedure *Trim_Cloaking_w* and show it in Algorithm 4.10. Figure 4.6(b) shows the TCRs for r_1 after applying the *Trim_Cloaking_w* algorithm with the additional QoS constraint $\Delta d_{max} = 2$ to each of the sub-windows of the result set shown in Figure 4.6(a). The cloaking region a from Figure 4.6(a) is filtered away, and TCR c from Figure 4.6(a) is reduced in window size and shown in Figure 4.6(b) as c'.

4.6.3.2 TDJ Cloaking

The TDJ operation fits naturally into Trajectory Cloaking in cases where QoS constraint must be considered, i.e. $\Delta d_{max} \neq NULL$ in the three tuple trajectory privacy policy $(w, k, \Delta d_{max})$, since it retrieves all the trajectory segments from the target dataset S, for all trajectories r in the query dataset R, given a distance constraint Δd and a time window w. Similar to TkNNJ, a TDJ operation can be performed on the trajectory dataset R against itself and the returned result set can be processed to suit the needs of trajectory cloaking. However, TDJ result set processing for the purpose of trajectory cloaking differs slightly from that of TkNNJ. We recall from Section 4.3 that the TDJ result set for a trajectory r is presented as the tuple $(r, \{(s, \{w'\})\})$, which returns the trajectory segments from the target dataset that satisfy the distance constraint together with the set of time windows in which the distance constraint is satisfied. Trajectory cloaking, on the other hand, requires that the TCRs be arranged in ascending order of the time windows, in each of which Window Trajectory k-Anonymity must hold. Consequently, the result set produced by TDJ must be processed to produce the TCS before it can be useful in assembling TCRs.

The *TDJ_post_process* procedure shown in Algorithm 4.11 illustrates the post

Algorithm 4.12: $Scan_Merge_Split((w', r'), tcs)$

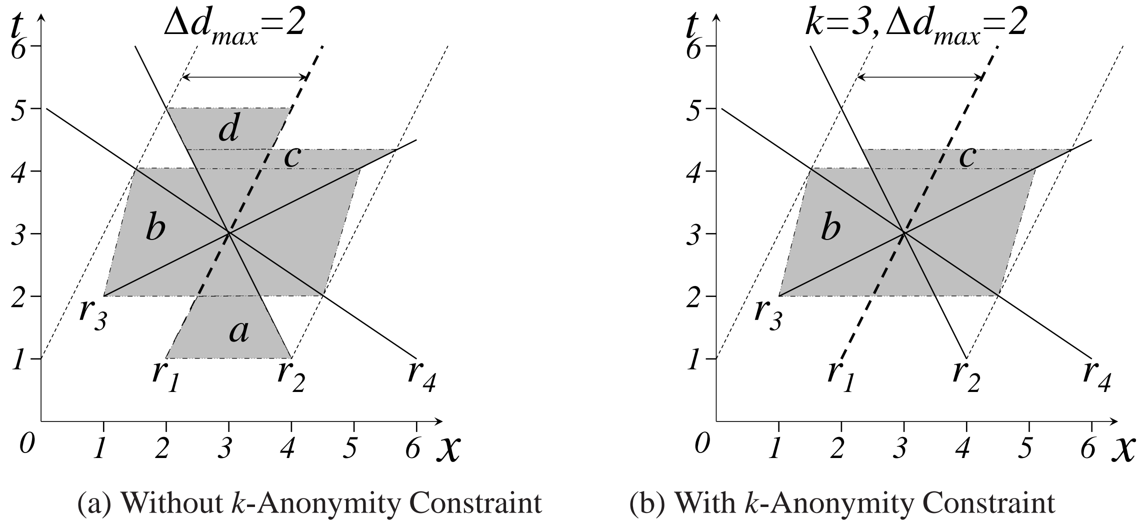
```
1  t1 ⇐ w'⊢; t2 ⇐ w'⊣;
2  (w'', {r''}, n) ⇐ scan_first(tcs);
3  if t2 < w'⊣ then insert((w', r', 1), tcs);
4  else
5      while w''⊣ < t1 do (w'', {r''}) ⇐ scan_next(tcs);
6      t' ⇐ t1;
7      while w''⊢ < t2 do
8          if w''⊢ > t' then
9              insert(([t', w''⊢), r', 1), tcs);
10             t' ⇐ w''⊢;
11         else
12             insert(([w''⊢, t'), {r''}, n), tcs);
13             insert(([t', w''⊣), {r''} ∪ {r'}, n + 1), tcs);
14             t' ⇐ w''⊣;
15         (w'', {r''}) ⇐ scan_next(tcs);
16     insert(([t', t2), {r'}, 1), tcs);
17 Return;
```

processing phase discussed above. To produce the TCS from the TDJ result set, the *TDJ_post_process* algorithm performs a linear scan of the TDJ result set, merges overlapping time windows, and splits non-overlapping ones, keeping track of the trajectory segments within the newly produced time windows at the same time. This sub-procedure is called Scan_Merge_Split and is presented in Algorithm 4.12.

We observe that the TCS for a trajectory obtained from the *TDJ_post_process* algorithm automatically guarantees the satisfaction of the QoS constraint, due to the nature of the TDJ operation. To fulfill the Trajectory k-Anonymity requirement, we simply need impose an additional cardinality constraint on the trajectory segment set $\{r'\}$ within each of the time windows, such that $n \geq k$, and filter out the time windows that do not meet this requirement.

The construction of TCRs from the TCS obtained from the *TDJ_post_process* algorithm is exactly the same as that discussed in the previous section and we omit the discussion to avoid redundancy.

Figure 4.7 shows the example TCRs constructed from the result set of the TDJ



(a) Without k-Anonymity Constraint (b) With k-Anonymity Constraint

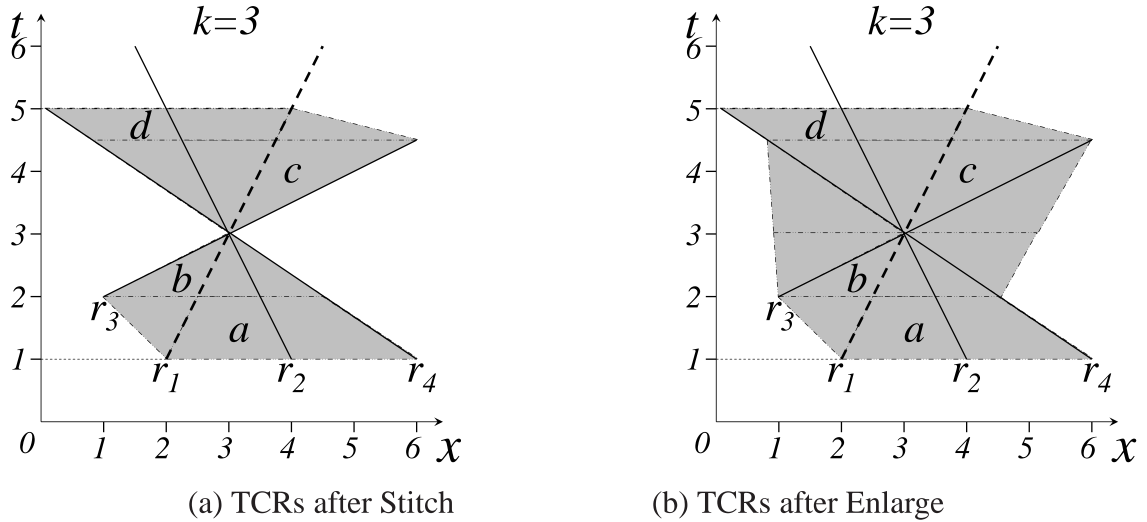
Figure 4.7: Trajectory k-Anonymity with TDJ

operation. Figure 4.7(a) shows the TCRs with only the QoS constraint $\Delta d_{max} = 2$, in which four TCRs, a, b, c, and d are presented. In Figure 4.7(b), however, the k-Anonymity constraint is considered with $k = 3$, and as a result TCRs a and d are filtered out because during these time windows the cardinality constraint on the TCS is not met.

4.6.4 Presenting *k*-Anonymous Trajectories

TCRs derived by the JiST cloaking procedures described in the above subsections satisfy the *Window Trajectory k-Anonymity* specified by privacy policy $(w, k, \Delta d_{max})$. However, several potential vulnerabilities exist. For example, in Figure 4.6(a) where TCRs without QoS constraint are shown, TCRs a and b, c and d are disconnected, which reveals the exact locations of all the trajectories in the TCRs at time window boundaries, as well as the trajectory segments bounding the TCRs. This vulnerability may be less pronounced in two-dimensional trajectories, due to higher uncertainty introduced by two-dimensional MBRs, but it poses a potential threat to the privacy of the disclosed trajectories nontheless. The second vulnerability is manifested at trajectory junction points, where all trajectory segments in the TCRs intersect at one single point, as is shown in Figure 4.6(b) at time 3, where the locations of all trajectory segments in TCRs b and c' are revealed.

To reduce these vulnerabilities, we introduce a two-phase heuristic for further



(a) TCRs after Stitch (b) TCRs after Enlarge

Figure 4.8: Presenting k-Anonymous Trajectories

processing of TCRs before disclosure: the *Stitch* phase, and the *Enlarge* phase.

Stitch: In the Stitch phase, we union the MBRs between time contiguous TCRs, e.g., the MBR of TCR *a* and the MBR of TCR *b* at time 2 in Figure 4.6(a), and form a larger MBR shared between the two TCRs, and in doing so stitch the two TCRs together. Figure 4.8(a) shows the TCRs for r_1 after the Stitch phase is applied to the TCRs shown in Figure 4.6.

Enlarge: In the Enlarge phase we take the TCRs produced by the Stitch phase, identify time contiguous TCRs that share point MBRs, connect the start MBR of the first TCR and the end MBR of the second TCR by their corresponding vertices, obtain intersection points between the line segments connecting the MBR vertices and the plane where the point MBR resides, and form a new MBR enclosing all intersection points and the point MBR itself, thus enlarging the point MBR to the new MBR. Figure 4.8(b) shows the TCRs for r_1 after the Enlarge phase is applied to the TCRs shown in Figure 4.8(a).

We note that the Stitch and Enlarge heuristic does not result in TCRs that violate the privacy policy, because both Stitch and Enlarge phases enlarge the TCRs within the QoS constraint. Proof of this is straightforward and is omitted.

Table 4.2: Experiment Parameters

Parameter	**Value**
Page Size	**4K**
Buffer Pool	**512K**
Dataset	50K, **100K**, 150K, 200K, 250K
Update Interval	**60**, 120, 180, 240
Index Lifetime	**120**
Horizon	**240**
Experiment Duration	**600**
Query Window	40, 80, **120**, 160, 200
Number of NNs	10, 20, **30**, 40, 50
Spatial Extent	1000× units
Join Distance	10, 50, **100**, 150, 200 units

4.7 Experimental Evaluations

In this section, we present results from a comprehensive experimental evaluation and evaluate the effectiveness of the time-parameterized join algorithms that we have.

4.7.1 Implementation Details

We implemented both the JiST index structure and the join algorithms on top the SHORE storage manager [22]. In the implementation of the JiST index, we followed the methods proposed in Chapter 2, with the extensions discussed in Section 4.5.

We show the experimental results in terms of execution time that consists of I/O time and CPU time. Results for queries presented in this section consist of both filtering step and refinement step, for both TDJ and TkNNJ query operations. For all the numbers shown in this section, five measurements were taken and the average of the middle three values is presented.

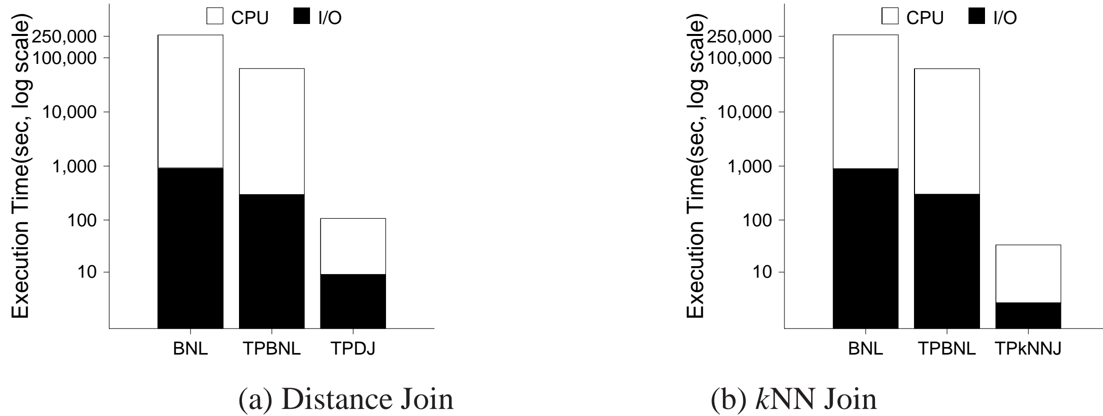
All experiments were conducted on a 2.0 GHz Intel Xeon dual processor workstation with 1 GB physical memory.

4.7.2 Experimental Settings

To keep the experiments manageable, we set the page size of the storage manager to $4K$ and the buffer pool to 128 pages. We have also experimented with a wide range of buffer pool and page sizes, the results are consistent with those presented in the following sections, and in the interest of space we have omitted these results.

Due to the lack of real world moving object data, we used the well-know GSTD data generator [107] to produce synthetic moving data. This data generator allows us to systematically generate data with various parameters and to explore the effect of these parameters on the performance of our algorithms. In most of the experiments we simulated moving objects within a $1,000 \times 1,000$ space domain, traveling with maximum speeds of 0.75, 1, or 3 per minute. One can imagine the unit of the space being a mile and the unit speed of the moving objects being miles per minute. Initially each object is randomlly assigned a location, a speed within one of the three speed groups, and a moving direction. After that, the objects keep moving at the assigned speed until the next update, at which time the location information is retained, but a different speed and direction are assigned randomly.

Using the GSTD data generator, we were able to adjust various parameter settings for the experiments. We generated synthetic datasets of various cardinalities and update intervals, as shown in Table 4.2. Table 4.2 also gives a list of query parameters used in the experiments. Parameter values shown in bold are default values. Since previous research [54] has studied the effect of the *Index Lifetime* parameter in the context of dual transformed index structure, we fix the value of this parameter at 120 time units to avoid redundant evaluations. Time windows of both historical and predictive join queries in the experiments are also generated randomly.



(a) Distance Join (b) *k*NN Join

Figure 4.9: Comparing with Naïve Algorithms: Historical

4.7.3 Comparing with Naïve Algorithms

In the first set of experiments we compare the dual index based join algorithms with the naïve BNL and TPBNL algorithms presented in Section 4.4.3.

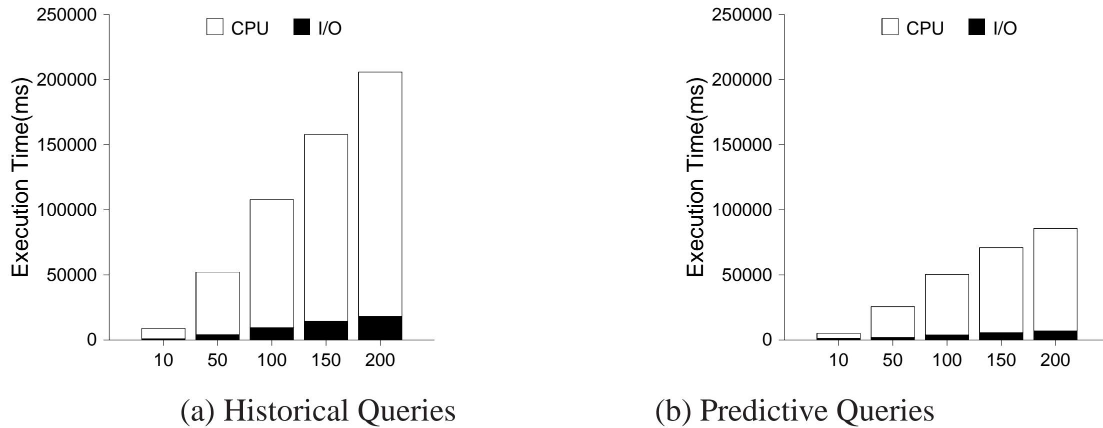
We performed the comparison on various combinations of parameter settings, for both historical and predictive queries. In the interest of space, we only show in Figure 4.9 the results for historical queries from running the experiments at default settings. Other results are consistent with those presented here.

As is expected, the dual index based algorithms outperform their naïve counterparts by orders of magnitude, due to the effectiveness of synchronized index traversal and intermediate filtering.

In the rest of this section we will focus mainly on the performance evaluation of the dual index based algorithms.

4.7.4 Evaluating Dual Index Based Algorithms

In this section we focus our discussion on the performance of the proposed dual index based join algorithms. We make assessment on TDJ and TP*k*NNJ algorithms individually on their distinct characteristics and jointly on their shared attributes.



(a) Historical Queries (b) Predictive Queries

Figure 4.10: TDJ−Effect of Join Distance Δd

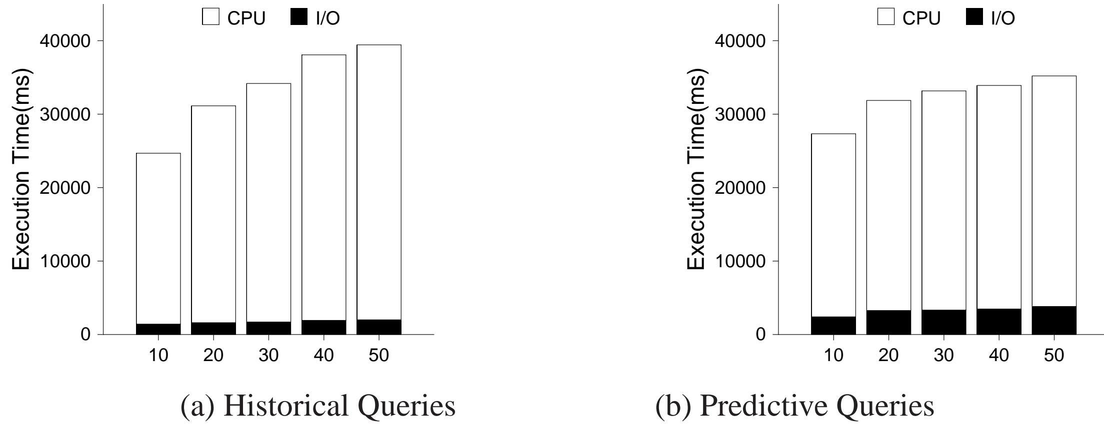
4.7.4.1 Effect of Δd in TDJ

There are three key factors to the TDJ operation: the Join Distance Δd, the Query Window w, and workload mix. Since the temporal parameter w and the effect of workload mix are common to both the TDJ operation and the TkNNJ operation, we leave them to a joint discussion in Sections 4.7.4.3 and 4.7.4.4.

To examine the effect of Δd on TDJ algorithm, we performed a series of experiments, in which a set of TDJ queries were issued on two datasets each with $100K$ moving objects. The TDJ parameter Δd was set to a fraction of the extent of the spatial domain, namely $1,000$. We varied Δd from 0.01, or 10 spatial units, to 0.2, or 200 spatial units. Both historical and predictive queries were examined, and the execution time per query is measured and shown on the y axis in Figure 4.10. A clear linear correlation between query execution time and Δd can be observed, for both historical and predictive queries, both in terms of I/O and CPU time.

4.7.4.2 Effect of k in TkNNJ

To assess the effect of k on the performance of TkNNJ algorithm, we conducted the following experiments. Fixing the datasets at $100K$, and the query window at 120 time units, we performed both historical and predictive TkNNJ queries, each with k value varying at 10, 20, 30, 40, and 50. The results are depicted in Figure 4.11, where the y axis



(a) Historical Queries (b) Predictive Queries

Figure 4.11: TkNNJ−Effect of k

shows the average execution time of a single query.

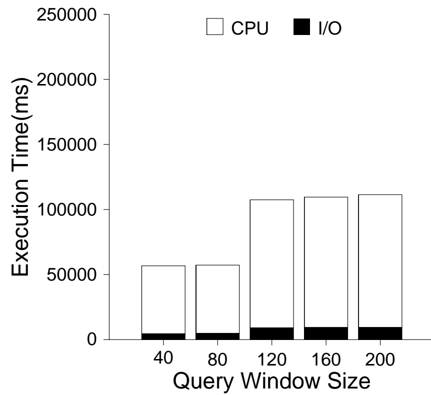
We draw two conclusions from Figure 4.11. For both historical and predictive queries, as the value of k increases, so does the execution time of TkNNJ query, albeit in a sub-linear fashion. This can be attributed to effectiveness of the filtering step in the $updateTPQ$ algorithm (Algorithm 4.3). The second observation to be made is that the sub-linear correlation between query execution time and value of k is slightly more pronounced in predictive queries, for reasons we will discuss in more detail in Section 4.7.4.5.

4.7.4.3 Effect of Query Window Size

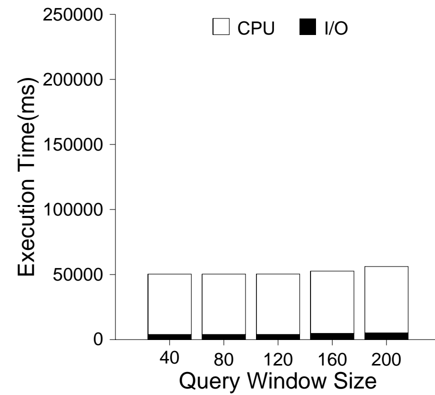
Both TDJ and TkNNJ queries take a common parameter, i.e., the query window. In this section we examine the effect of the query window size on query performance.

The default $100K$ dataset size is used in this experiment. We varied the query window size as shown in Table 4.2 for both TDJ and TkNNJ queries, each in turn consist of both historical and predictive queries. The results are shown in Figures 4.12 and 4.13.

As can be observed from these figures, larger query window results in longer query execution time for both TDJ and TkNNJ queries, in both historical and predictive settings. The reason for this trend is intuitive: the larger the query window, the more objects are likely to be active within that time window, and thus the more computation is likely to be

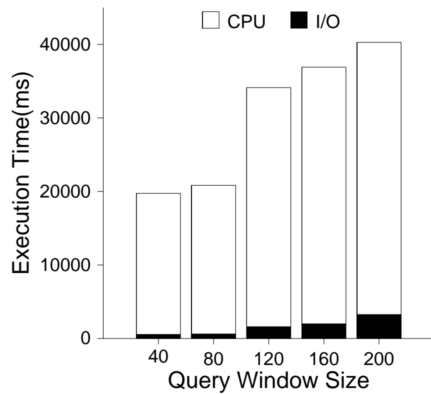


(a) Historical Queries

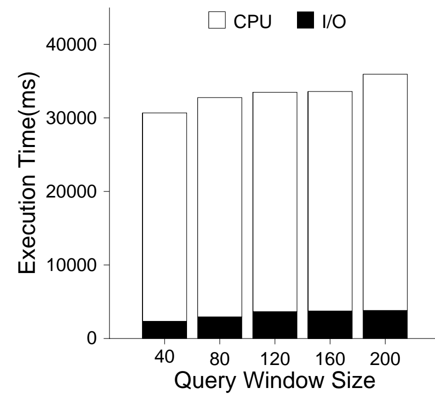


(b) Predictive Queries

Figure 4.12: Effect of Query Window Size *w*: TDJ



(a) Historical Queries



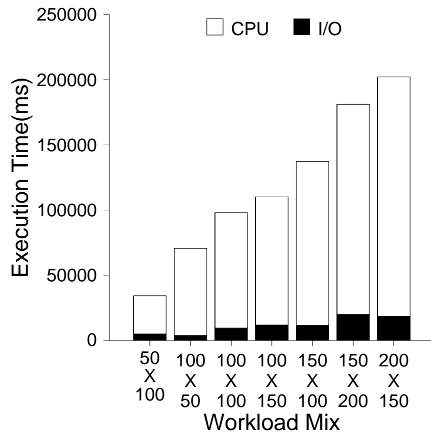
(b) Predictive Queries

Figure 4.13: Effect of Query Window Size *w*: TkNNJ

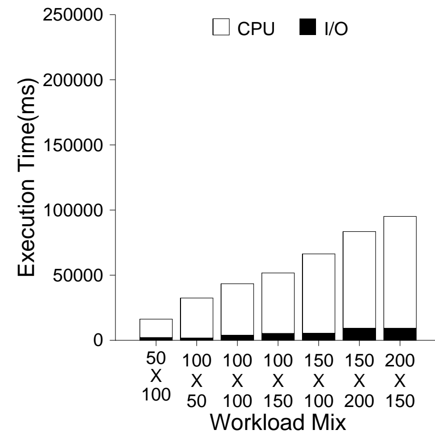
incurred by the queries.

Furthermore, one can also observe that the extension of the query window has a more dramatic effect on TkNNJ than on TDJ. This phenomenon can be attributed to the fact that TkNNJ query filtering and refinement procedures are more complicated, as can be inferred from the TkNNJ algorithms presented in Section 4.4. The shrinking or expansion in query window size inevitably induces the decrease or increase in the number of WPQs that need to be processed during the query process. This results in a more profound effect of query window size on TkNNJ query than on TDJ query.

A third observation to be made here is the jump in execution time that corresponds to the change in window size from 80 to 120 for both TDJ (Figure 4.12(a)) and TkNNJ



(a) Historical Queries



(b) Predictive Queries

Figure 4.14: Effect of Workload Mix: TDJ

(Figure 4.13(a)) query operations. This jump is specific to historical queries, and is due to the fact that the lifetime of the base indexes in JiST is set to 120 time units, and the randomly generated historical queries mostly have an upper bound very close to the end of an index lifetime. For this reason, queries with window size 40 or 80 time units span only one single historical index structure most of the time, whereas queries with window size greater than 120 time units almost always straddle two index structures. This results in approximately 2*X* increase in query execution time when going from a window size of 40 or 80 to a window size of 120 and above, as can be observed from Figures 4.12(a) and 4.13(a).

However, the effect observed above is not applicable in the case of predictive queries, for reasons we will further discuss in Sub-section 4.7.4.5.

4.7.4.4 Effect of Workload Mix

In this section we examine the effect of various workload mixes for both TDJ and TkNNJ operations.

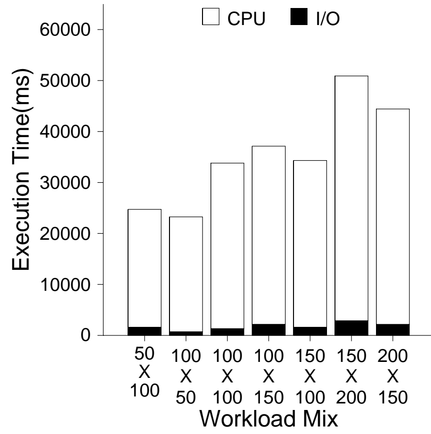
We conducted two sets of experiments where we gradually increase the size of both the outer relation **R** and the inner relation **S**. However, in the first set of experiments we keep the size of the inner relation to be either the same as the outer relation (for the

default 100K dataset), or slightly larger. In the second set of experiments we reverse the trend and keep the size of the inner relation to be slightly smaller. We experimented with a wide range of dataset sizes and obtained consistent results. In the interest of space, we presents the results for the mix of four dataset cardinalities: 50K, 100K, 150K, and 200K in Figures 4.14 and 4.15.

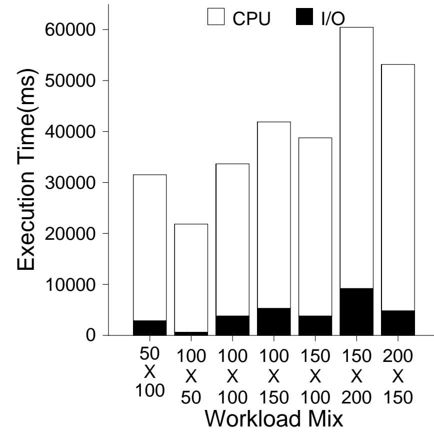
We observe from Figures 4.14 and 4.15 that, for both TDJ and TkNNJ operations, as the size of the outer relation increases, both CPU and I/O times for the operation increase in a linear fashion (observe the bars on the 50×100, 100×100, and 150×100 markers). Note that in Figures 4.14 and 4.15 the numbers on the top of the markers indicate the sizes of the outer relations. Similar trend also holds for the size of the inner relation (the bars on the 100×50, 100×100, and 100×150 markers). This is so because for both TDJ and TkNNJ operations, the time required for distance computation and result retrieval for both filtering and refinement stages is positively correlated with the cardinality of both inner and outer relations.

We would also like to point out that, from the results shown in Figure 4.14, for TDJ queries, the change in the size of the outer relation has a more pronounced impact on the execution time than that of the inner relation. Compare the bars corresponding to the 50×100 markers against those corresponding to the 100×50 markers in Figure 4.14, as well as those corresponding to 100×150 against 150×100, and 150×200 against 200×150. Observe that in all these cases, despite the decrease in size of the inner relation, TDJ query execution time increases with the size of the outer relation. However, one may also notice that the query time increase is mainly due to the increase in CPU time. This can be attributed to the fact that the refinement stage in the TDJ query algorithm is largely affected by the number of trajectories in the outer relation, and since this stage is executed in memory, only CPU time is affected.

On the other hand, corresponding bars in Figure 4.15 show a completely different trend for TkNNJ queries. Due to the asymmetric nature of TkNNJ queries, the size of the



(a) Historical Queries



(b) Predictive Queries

Figure 4.15: Effect of Workload Mix: TkNNJ

inner relation plays a much more important role in the effectiveness of the query, both in terms of CPU and I/O time. One can infer from the TkNNJ algorithms presented in Section 4.4 that both CPU and I/O costs are linearly correlated with the sizes of the inner and outer relations during the filtering stage. However, during the refinement stage, the split operations on the WPQs are again linearly correlated with the number of candidates in the candidate sets, which is in turn linearly correlated with the size of the inner relation. This results in a near-quadratic effect, which is manifested in Figure 4.15 to some extent.

4.7.4.5 Historical vs. Predictive Queries

Comparing all figures in this section presenting historical query performance and those presenting predictive query performance, we make the following observation with respect to the performance of historical and predictive TDJ queries: processing predictive TDJ queries requires significantly less time ($< 0.5X$) than their historical counterparts, given exactly the same query parameters.

The rationale for this observation is straightforward. Historical data contains complete information about the past trajectories of moving objects, and as such, Validity Intervals of individual objects are very likely to be segmented and often much shorter than the lifetime of one single index structure, depending on the length of the update interval. This results

in a significant number of TPD computations. On the other hand, future positions of moving objects can only be inferred based on their current position and velocity, and thus follow the pattern of a straight line. Consequently, a single TPD function is applicable for the entire query duration, therefore incurring much less computation overhead.

The above mentioned effect holds for CPU cost of TkNNJ queries, as can be observed from all historical TkNNJ query performance figures presented in this section. The reason for this is similar to that discussed above. However, specific to TkNNJ queries, we observe that almost in all cases the I/O cost of predictive queries turns out to be higher than that of their historical counterparts. This is due to the diminished effectiveness of the filtering step because of the expansion of TBRs over a long period of time. In processing the predictive queries, the computation of TPD is based on the predicted TBRs, which expand rapidly as times progresses, inducing increased overlapping that reduces the effectiveness of the pruning during the filtering stage. This results in an increased number of node expansions that in turn incur more disk I/Os.

4.8 Conclusions and Future Work

In this chapter we have introduced JiST, a general framework for Trajectory Join operations including Trajectory Distance Joins (TDJ) and Trajectory *k* Nearest Neighbors Joins (TkNNJ). In addition, we have presented a set of table scan based algorithms as well as index based algorithms that take advantage of dual transformed index structures in evaluating the join operations introduced in this chapter. We have applied the JiST framework operations to develop novel techniques for trajectory privacy preservation. Finally, we have evaluated the performance of the proposed trajectory join algorithms through exhaustive experiments and demonstrated the effectiveness of these methods.

To the best of our knowledge, JiST is the first comprehensive framework for complex spatio-temporal join operations between moving object trajectories, and its application in trajectory privacy preservation is unprecedented. JiST will serve as the basis for many

future research efforts, including design of ad hoc query operations, complex query evaluation, query size estimation, and optimization methods for emerging location-based applications, among which location privacy preservation is a promising direction. For simplicity, we will use the term *Location Privacy* to incorporate the meanings of both location privacy and trajectory privacy.

Location privacy is a rapidly growing research area that offers numerous exciting opportunities. Overall, there are two main factors to consider when developing location privacy preservation techniques, i.e., the efficiency and effectiveness of the technique, and the quality of service of the LBS applications using the technique. We provide some insight on how we can exploit the comprehensiveness and versatility provided by the JiST framework to explore these opportunities both in developing efficient and effective techniques and in preserving quality of service.

In the time domain, since the JiST index structure spans the entire lifetime of all trajectory data, it is capable of accommodating anonymous access to historical, current, and predicted future trajectory data, both at precise time points and within a time duration. The collection of spatio-temporal query operations that can be derived from the JiST framework provide a rich choice of efficient query processing primitives for developing privacy preservation techniques. In this chapter we have discussed methods for preserving privacy on historical trajectories during a given time window in the past. We note that the extension of the proposed algorithms to current and predicted future trajectories can be easily derived, due to the dual transform nature of the underlying JiST index structure, as we discussed in Section 4.4. It is also straightforward to shrink the time window in the proposed algorithms to achieve location privacy preservation for a single time point, be it in the past, the present, or the near future.

In the spatial domain, we consider both the effectiveness aspect and the QoS aspect of the location privacy preservation techniques.

The effectiveness of a privacy preservation technique refers to the strength of privacy

provided by the technique, namely, how difficult it is to infer the exact location information of objects given the information provided by the technique. Here we extend the notion of *Reciprocity* proposed by Ghinita et. al. [37] in the context of location k-Anonymity preservation. The k-Anonymity reciprocity property dictates that all objects within a k-anonymous region share exactly the same k-anonymous region. In other words, the k-anonymous regions satisfying the reciprocity property do not overlap, which is essentially a form of space partitioning. Since location privacy preservation techniques are not only limited to k-Anonymity, we generalize the notion of the reciprocity property and define it as the property that *all objects within a privacy preserving region with respect to certain privacy policy, share exactly the same privacy preserving region, and no other*. We call techniques that observe this property *strong privacy techniques*, and call those that do not observe this property *weak privacy techniques*. We note that the methods we have provided in this chapter are weak privacy techniques since they address the privacy preservation of individual trajectories, and it is highly likely that the TCRs of two trajectories will have incomplete overlap. However, we argue that the JiST framework provides natural mechanisms for developing strong privacy techniques. The rationale is that the JiST index is built on top of STRIPES, which is essentially a space partitioning index structure. Furthermore, an important property of the STRIPES index structure is that objects with similar movement patterns in physical space also tend to be close to each other on the index, which inherently offers the opportunities of object clustering and space partitioning techniques.

In this chapter we have provided a distance constraint Δd_{max} as the QoS criterion in trajectory privacy policy specification. However, we observe that the JiST framework is also capable of addressing the more commonly adopted rectangular area constraints, which are essentially in the form of time-parameterized range queries and have been discussed in detail in Chapter 2 under the setting of predictive query processing. Since the JiST index consists of slightly modified STRIPES structures, the extension of the STRIPES query

algorithms to the JiST index is fairly straightforward.

CHAPTER 5

CONCLUSIONS

In this thesis we have developed a comprehensive and unified framework for efficient access methods and query operations in spatio-temporal databases.

In Chapter 2 we have presented a new indexing structure called STRIPES for indexing and answering queries on predicted positions in moving object databases. This new indexing structure draws inspiration from earlier largely theoretical work in this area, advocating the use of dual transformation for indexing such data sets. The STRIPES index leverages these dual transformation techniques and uses a disjoint regular partitioning technique to efficiently index the points in the dual transformed space. The STRIPES index can support all the types of commonly used predictive queries [72], which include time-slice, window, and moving queries. We have compared the performance of STRIPES with the most efficient predictive indexing structure, the TPR*-tree [105]. Our comprehensive experimental evaluations demonstrate that STRIPES outperforms the leading competitive index method, namely the TPR*-tree index, for both updates and queries; updates are often more than an order of magnitude faster using STRIPES, and queries are often faster by a factor of 4x. These differences can be seen in both the I/O and the CPU costs. Consequently, STRIPES is an extremely efficient and practical indexing structure for evaluating predictive queries.

In Chapter 3 we have presented a new metric, called NXNDIST, and have shown that this metric is much more effective for pruning ANN computation than previously proposed methods. We have also explored the properties of this metric, and have presented an

efficient $O(D)$ algorithm for computing this metric, where D is the data dimensionality. We have also explored a family of index based methods for computing ANN queries. In addition, for ANN computation, we have shown that traversing the index trees using a depth-first paradigm, and using a bi-directional expansion of candidate search nodes is the most efficient strategy. With the application of NXNDIST, we have also shown how to extend our solution to efficiently evaluate the more general AkNN operation.

In Chapter 3 we have also shown that for ANN queries, using a quadtree index enhanced with MBR keys for the internal nodes, is a more efficient indexing structure than the commonly used R*-tree index. Overall the methods that we have presented generally result in significant speed-up of at least 2X for ANN computation, and over an order of magnitude for AkNN computation over the previous best algorithms (BNN [116] and GORDER [112]), for both low and high-dimensional datasets.

In Chapter 4 we have introduced JiST, a general framework for Trajectory Join operations including Trajectory Distance Joins (TDJ) and Trajectory k Nearest Neighbors Joins (TkNNJ). In addition, we have presented a set of table scan based algorithms as well as index based algorithms that take advantage of dual transformed index structures in evaluating the join operations introduced in this chapter. We have applied the JiST framework operations to develop novel techniques for trajectory privacy preservation. Finally, we have evaluated the performance of the proposed trajectory join algorithms through exhaustive experiments and demonstrated the effectiveness of these methods.

To the best of our knowledge, JiST is the first comprehensive framework for complex spatio-temporal join operations between moving object trajectories, and its application in trajectory privacy preservation is unprecedented. JiST will serve as the basis for many future research efforts, including designing of ad hoc query operations, complex query evaluation, query size estimation, and optimization methods for emerging location based applications.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Reverse 911. Website. http://www.reverse911.com/.

[2] The UCI Knowledge Discovery in Databases Archive. Website. http://kdd.ics.uci.edu/.

[3] Twin Astrographic Catalog Version 2 (TAC 2.0). Website, 1999. http://ad.usno.navy.mil/tac/.

[4] Automated Phone System Warns San Diego. Website, 2007. http://query.nytimes.com/gst/fullpage.html?res=9E01E2DC1E39F937A15753C1A9619C8B63.

[5] EMERGENCY NOTIFICATION TECHNOLOGY AIDS IN EVACUATION OF MORE THAN 500,000 SOUTHERN CALIFORNIA RESIDENTS. Website, 2007. http://www.reverse911.com/sites/www.reverse911.com/files/San_Diego_Wild_Fires.pdf.

[6] P. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *PODS*, pages 175–186, 2000.

[7] P. Bahl and V. N. Padmanabhan. RADAR: An In-building RF-based User Location and Tracking System. In *INFOCOM (2)*, pages 775–784, 2000.

[8] P. Bakalov, M. Hadjieleftheriou, E. J. Keogh, and V. J. Tsotras. Efficient Trajectory Joins Using Symbolic Representations. In *MDM*, pages 86–93. ACM, 2005.

[9] B. Bamba and L. Liu. PRIVACYGRID: Supporting Anonymous Location Queries in Mobile Environments. Technical report, Georgia Institute of Technology, 2007.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, 1990.

[11] R. Bellman. *Adaptive Control Processes*. Princeton University Press, Princeton, NJ, 1961.

[12] S. Berchtold, D. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *VLDB*, pages 28–39, 1996.

[13] A. R. Beresford and F. Stajano. Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, 2(1):46–55, 2003.

[14] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is “Nearest Neighbor” Meaningful? In *ICDT*, pages 217–235, 1999.

[15] C. Böhm, S. Berchtold, and D. A. Keim. Searching in High–Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[16] C. Böhm and F. Krebs. High Performance Data Mining Using the Nearest Neighbor Join. In *IEEE International Conference on Data Mining(ICDM)*, pages 43–50, 2002.

[17] C. Böhm and F. Krebs. Supporting KDD Applications by the k-Nearest Neighbor Join. In *DEXA*, pages 504–516, 2003.

[18] C. Böhm and F. Krebs. The k-Nearest Neighbor Join: Turbo Charging the KDD Process. *KAIS*, 6(6):728–749, 2004.

[19] B. Braunmüller, M. Ester, H.-P. Kriegel, and J. Sander. Efficiently Supporting Multiple Similarity Queries for Mining in Metric Databases. In *ICDE*, pages 256–267, 2000.

[20] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD*, pages 237–246, 1993.

[21] M. Cai, D. Keshwani, and P. Revesz. Parametric Rectangles: A Model for Querying and Animation of Spatiotemporal Satabases. In *EDBT*, pages 430–444, 2000.

[22] M. Carey and et al. Shoring Up Persistent Applications. In *SIGMOD*, pages 383–394, 1994.

[23] V. Chakka, A. Everspaugh, and J. Patel. Indexing Large Trajectory Data Sets with SETI. In *CIDR*, pages 164–175, 2003.

[24] Y. Chen and J. M. Patel. Efficient Evaluation of All-Nearest-Neighbor Queries. In *ICDE*, pages 1056–1065. IEEE, 2007.

[25] H. Chon, D. Agrawal, and A. Abbadi. Storage and Retrieval of Moving Objects. In *MDM*, pages 173–184, 2001.

[26] C.-Y. Chow, M. F. Mokbel, and X. Liu. A Peer-to-Peer Spatial Cloaking Algorithm for Anonymous Location-Based Service. In *GIS*, pages 171–178, 2006.

[27] K. L. Clarkson. Fast Algorithms for the All Nearest Neighbors Problem. In *24th Ann. IEEE Sympos. on the Found. Comput. Sci.*, pages 226–232, 1983.

[28] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *SIGMOD*, pages 189–200, 2000.

[29] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Algorithms for Processing K-Closest-Pair Queries in Spatial Databases. *TKDE*, 49(1):67–104, 2004.

[30] *FCC Wireless 911 Requirements*. Federal Communications Commission, 2001. http://www.fcc.gov/e911/factsheet_requirements_012001.pdf.

[31] D. J. Eisenstein and P. Hut. HOP: A New Group-finding Algorithm for N-Body Simulations. *The Astrophysical Journal*, 498:137–142, 1998.

[32] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join Operations in Temporal Databases. *VLDB J*, 14(1):2–29, 2005.

[33] I. Gargantini. An Effective Way to Represent Quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[34] B. Gedik and L. Liu. Location Privacy in Mobile Systems: A Personalized Anonymization Model. In *ICDCS*, pages 620–629, Washington, DC, USA, 2005. IEEE Computer Society.

[35] B. G. Gedik and L. Liu. A Customizable k-Anonymity Model for Protecting Location Privacy. Technical report, Georgia Institute of Technology, 2004.

[36] I. A. Getting. The Global Positioning System. *IEEE Spectrum*, 30(12):36–47, 1993.

[37] G. Ghinita, P. Kalnis, and S. Skiadopoulos. PRIVÉ: Anonymous Location-Based Queries in Distributed Mobile Systems. In *WWW*, pages 371–380. ACM, 2007.

[38] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-Memory Computational Geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[39] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *MobiSys*. USENIX, 2003.

[40] M. Gruteser and X. Liu. Protecting Privacy in Continuous Location-Tracking Applications. *IEEE Security & Privacy*, 2(2):28–34, 2004.

[41] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. Technical Report LBL-28587, Lawrence Berkeley Laboratories, 1989.

[42] S. Guo, Z. Huang, H. V. Jagadish, B. C. Ooi, and Z. Zhang. Relaxed Space Bounding for Moving Objects: A Case for the Buddy Tree. *SIGMOD Record*, 35(4):24–29, 2006.

[43] R. Guting, M. Bohlen, M. Erwig, and et al. A Foundation for Representing and Querying Moving Objects. *ACM TODS*, 25(1):1–42, 2000.

[44] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*, pages 47–57, 1984.

[45] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. In *Mobile Computing and Networking*, pages 59–68, Seattle, WA, USA, 1999.

[46] G. R. Hjaltason and H. Samet. Incremental Distance Join Algorithms for Spatial Databases. In *SIGMOD*, pages 237–248, 1998.

[47] G. R. Hjaltason and H. Samet. Speeding up Construction of PMR Quadtree-based Spatial Indexes. *VLDB Journal*, 11(2):109–137, 2002.

[48] E. G. Hoel and H. Samet. Benchmarking Spatial Join Operations with Spatial Output. In *VLDB*, pages 606–618, 1995.

[49] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations. In *VLDB*, pages 396–405, 1997.

[50] G. S. Iwerks, H. Samet, and K. Smith. Maintenance of Spatial Semijoin Queries on Moving Points. In *VLDB*, pages 828–839, 2004.

[51] H. Jagadish. On Indexing Line Segments. In *VLDB*, pages 614–625, 1990.

[52] A. K. Jain, M. N. Murty, and P. J. Flynn. Data Clustering: A Review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[53] R. Jarvis and E. Patrick. Clustering Using a Similarity Measure Based on Shared Near Neighbors. 22:1025–1034, 1973.

[54] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, pages 768–779, 2004.

[55] S.-H. Jeong, N. W. Paton, A. A. A. Fernandes, and T. Griffitths. An Experimental Performance Evaluation of Spatio-Temporal Join Strategies. *TGIS*, 9(2), 2004.

[56] S. C. Johnson. Hierarchical Clustering Schemes. *Psychometrika*, 2:241–254, 1967.

[57] H. Kido, Y. Yanagisawa, and T. Satoh. An Anonymous Communication Technique using Dummies for Location-based Services. In *ICPS*, 2005.

[58] S. Koenig and Y. Smirnov. Graph Learning with a Nearest Neighbor Approach. In *Proceedings of the Conference on Computational Learning Theory*, pages 19–28, 1996.

[59] G. Kollios, D. Gunopulos, and V. Tsotras. On Indexing Mobile Objects. In *PODS*, pages 261–272, 1999.

[60] R. K. V. Kothuri, S. Ravada, and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison Using GIS Data. In *SIGMOD*, pages 546–557, 2002.

[61] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *MDM*, pages 113–120, 2002.

[62] M.-L. Lee, W. Hsu, C. Jensen, and et al. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, pages 608–619, 2003.

[63] T. Y. C. Leung and R. R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In *Temporal Databases*, pages 329–355. 1993.

[64] S. Leutenegger and M. Lopez. The Effect of Buffering on the Performance of R-trees. In *TKDE*, pages 33–44, 2000.

[65] D. Lin, C. S. Jensen, B. C. Ooi, and S. Saltenis. Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects. In *MDM*, pages 59–66. ACM, 2005.

[66] J. Lin, E. J. Keogh, S. Lonardi, and B. Y. chi Chiu. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In *DMKD*, pages 2–11, 2003.

[67] L. Liu. From Data Privacy to Location Privacy: Models and Algorithms. In *VLDB*, pages 1429–1430. ACM, 2007.

[68] M.-L. Lo and C. V. Ravishankar. Spatial Hash-Joins. In *SIGMOD*, pages 247–258, 1996.

[69] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-Diversity: Privacy Beyond k-Anonymity. In *ICDE*, page 24, 2006.

[70] S. Šaltenis and C. Jensen. Indexing of Moving Objects for Location-Based Service. In *ICDE*, pages 463–472, 2002.

[71] S. Šaltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, pages 331–342, 2000.

[72] S. Šaltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the Positions of Continuously Moving Objects. http://www.cs.auc.dk/˜simas, 2003. TPR-tree Source Code.

[73] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The New Casper: Query Processing for Location Services without Compromising Privacy. In *VLDB*, pages 763–774. ACM, 2006.

[74] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[75] M. Nascimento, J. Silva, and Y. Theodoridis. Evaluation of Access Structures for Discretely Moving Points. In *STDBM*, pages 171–188, 1999.

[76] M. A. Nascimento and J. R. O. Silva. Towards Historical R-trees. In *ACM-SAC*, pages 235–240, 1998.

[77] J. Ni and C. V. Ravishankar. Indexing Spatio-Temporal Trajectories with Efficient Polynomial Approximations. *IEEE Trans. Knowl. Data Eng.*, 19(5):663–678, 2007.

[78] R. Nock, M. Sebban, and D. Bernard. A simple locally adaptive nearest neighbor rule with application to pollution forecasting. *Internal Journal of Pattern Recognition and Artificial Intelligence*, 17(8):1–14, 2003.

[79] M. Pallavicini, C. Patrignani, M. Pontil, and A. Verri. The Nearest-Neighbor Technique for Particle Identification. *Nucl. Instr. and Meth.*, 405:133–138, 1998.

[80] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group Nearest Neighbor Queries. In *ICDE*, pages 301–312, 2004.

[81] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing Spatio-temporal Data Warehouses. In *ICDE*, pages 166–175, 2002.

[82] A. Papadopoulos and Y. Manolopoulos. Performance of Nearest Neighbor Queries in R-Trees. In *ICDT*, pages 394–408, 1997.

[83] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *SIGMOD*, pages 635–646. ACM Press, 2004.

[84] J. M. Patel and D. J. DeWitt. Partition Based Spatial-merge Join. In *SIGMOD*, pages 259–270, 1996.

[85] D. Pfoser, C. Jensen, and Y. Theodoidis. Novel Approaches to the Indexing of Moving Objects Trajectories. In *VLDB*, pages 395–406, 2000.

[86] E. Pitoura and G. Samaras. Locating Objects in Mobile Computing. *IEEE TKDE*, 13(4):571–592, 2001.

[87] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying Mobile Objects in Spatio-temporal Databases. In *SSTD*, pages 59–78, 2001.

[88] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[89] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-support System. In *Mobile Computing and Networking*, pages 32–43, Boston, MA, USA, 2000.

[90] C. Procopiuc, P. Agarwal, and S. Har-Peled. STAR-Tree: An Efficient Self-adjusting Index for Moving Objects. In *Algorithm Engineering and Experiments*, pages 178–193, 2002.

[91] S. Ramaswamy and T. Suel. I/O-Efficient Join Algorithms for Temporal, Spatial, and Constraint Databases. Technical report, Bell Labs, 1996.

[92] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD*, pages 71–79, 1995.

[93] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[94] Satyendra P. Rana and Farshad Fotouhi. Efficient Processing of Time-Joins in Temporal Data Bases. In *DASFAA*, volume 4 of *Advanced Database Research and Development Series*, pages 427–432. World Scientific, 1993.

[95] H. Shin, B. Moon, and S. Lee. Adaptive Multi-Stage Distance Join Processing. In *SIGMOD*, pages 343–354, 2000.

[96] D. Son and R. Elmasri. Efficient Temporal Join Processing Using Time Index. In *Proceedings: Eighth International Conference on Scientific and Statistical Database Systems*, pages 252–261. IEEE Computer Society Press, 1996.

[97] Z. Song and N. Roussopoulos. Hashing Moving Objects. In *MDM*, pages 161–172, 2001.

[98] Z. Song and N. Roussopoulos. SEB-tree: An Approach to Index Continuously Moving Objects. In *MDM*, pages 340–344, 2003.

[99] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *ICDE*, pages 282–292, 1994.

[100] J. Sun, Y. Tao, D. Papadias, and G. Kollios. Spatio-temporal Join Selectivity. *Information Systems*, 2006.

[101] L. Sweene. Achieving k-Anonymity Privacy Protection Using Generalization and Suppression. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):571–588, 2002.

[102] L. Sweene. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[103] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In *SIGMOD*, pages 611–622, 2004.

[104] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB Journal*, pages 431–440, 2001.

[105] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, pages 790–801, 2003.

[106] J. Tayeb, O. Ulusoy, and O. Wolfson. A Quadtree-based Dynamic Attribute Indexing Method. *Computer Journal*, 41(3):185–200, 1998.

[107] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. *Lecture Notes in Computer Science*, 1651:147–164, 1999.

[108] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-Temporal Indexing for Large Multimedia Application. In *ICMCS*, pages 441–448, 1996.

[109] Y.-R. Wang, S.-J. Horng, and C.-H. Wu. Efficient Algorithms for the All Nearest Neighbor and Closest Pair Problems on the Linear Array with a Reconfigurable Pipelined Bus System. *PDIS*, 16(3):193–206, 2005.

[110] A. Ward, A. Jones, and A. Hopper. A New Location Technique for the Active Office. *IEEE Personnel Communications*, 4(5):42–47, 1997.

[111] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *SSDBM*, pages 111–122, Capri, Italy, 1998.

[112] C. Xia, H. Lu, B. C. Ooi, and J. Hu. GORDER: An Efficient Method for KNN Join Processing. In *VLDB*, pages 756–767, 2004.

[113] X. Xu, J. Han, and W. Lu. RT-tree: An Improved R-Tree Index Structure for Spatio-Temporal Databases. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, pages 1040–1049, 1990.

[114] J. S. Yoo, S. Shekhar, and M. Celik. A Join-less Approach for Co-location Pattern Mining: A Summary of Results. In *IEEE International Conference on Data Mining(ICDM)*, pages 813–816, 2005.

[115] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient Temporal Join Processing Using Indices. In *ICDE*, page 103. IEEE Computer Society, 2002.

[116] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-Nearest-Neighbors Queries in Spatial Databases. In *SSDBM*, pages 297–306, 2004.

[117] M. Zhu, D. Papadias, J. Zhang, and D. L. Lee. Top-k Spatial Joins. *TKDE*, 17(4):567–579, 2005.

[118] T. Zurek. *Optimization of Partitioned Temporal Joins*. PhD thesis, University of Edinburgh, 1997.