Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Signature:

Xiaofeng Xu	Date

Indexing Moving Objects for Predictive Spatio-Temporal Queries

by

Xiaofeng Xu
Doctor of Philosophy

Computer Science and Informatics

Li Xiong, Ph.D.
Advisor

Vaidy Sunderam, Ph.D.
Advisor

Mohamed F. Mokbel, Ph.D.
Committee member

Ymir Vigfusson, Ph.D.
Committee member

Accepted:

Lisa A. Tedesco, Ph.D.
Dean of the James T. Laney School of Graduate Studies

Date

Indexing Moving Objects for Predictive Spatio-Temporal Queries

by

Xiaofeng Xu
B.S., Sun Yat-sen University, 2011

Advisor: Li Xiong, Ph.D.
Advisor: Vaidy Sunderam, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2016

Abstract

Indexing Moving Objects for Predictive Spatio-Temporal Queries
By Xiaofeng Xu

The rapid development of positioning techniques has enabled information to be widely collected on continuously moving objects, such as vehicles and mobile device users. Since moving object data is large and updates frequently, database systems with spatio-temporal indexes that support massive updates and predictive spatio-temporal queries are essential for modern location-based services. In this dissertation, I present novel approaches that augment existing tree-based and grid-based indexes for moving object databases with velocity information and prove that these approaches can significantly improve query performance with comparable update performance in both in-disk and in-memory scenarios. Predictive range query, which retrieves objects in a certain spatial region at some future time, is the most motivating type of spatio-temporal queries in real world location-based services. Different from predicting future location of a single moving object, performing predictive range queries over large moving object databases incurs much heavier computational burden, which makes efficiency as important as accuracy for real-time spatio-temporal enquiries. Motion functions, which predict future object locations based on some analytic functions, can efficiently process short-term predictive range queries but are not suitable for long-term predictions since motions of the moving objects might change over time. Other prediction functions such as trajectory patterns and statistical graphic models are more accurate but less efficient. In this dissertation, I also present a pruning mechanism that improve the performance for long-term predictive range queries based on (high-order) Markov chain models learned from historical trajectories. The key to our approach is to devise compressed representations for sparse multi-dimensional matrices, and leverage efficient algorithms for matrix computations. We conduct experiments on both simulated and real world datasets to demonstrate that our methods gain significant improvements over other existing methods.

Indexing Moving Objects for Predictive Spatio-Temporal Queries

by

Xiaofeng Xu
B.S., Sun Yat-sen University, 2011

Advisor: Li Xiong, Ph.D.
Advisor: Vaidy Sunderam, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2016

Acknowledgments

First of all, I would like to thank my advisor Prof. Li Xiong, for her support and guidance of my postgraduate study and research. Li generously offered her inspiration and knowledge for helping me in many aspect of doing researches from seeking research topics to formalizing research problems. She also encouraged me to attend academic conferences and to seek industrial research opportunities, so that I can exchange with other researchers and touch the cutting-edge technologies. I would like to thank my co-advisor Prof. Vaidy Sunderam, for his insightful suggestions on my research. Discussing with Li and Vaidy was delightful and productive, from which I learned that perseverance, criticism and enthusiasm are the keys to become a good researcher. Thanks to my dear advisors for working with me day and night to polish the research ideas and papers.

I would also like to thank my thesis committee members, Prof. Ymir Vigfusson and Prof. Mohamed Mokbel, for their insightful suggestions and detailed comments on my dissertation. Thanks to Prof. Mohamed Mokbel for coming from Minnesota for my thesis proposal.

Thanks to Prof. James Nagy and Prof. Fusheng Wang for directing my early research projects. Thanks to the faculties, Prof. Eugene Agichtein, Prof. Michele Benzi, Prof. Michelangelo Grigni, Prof. Cengiz Gunay, Prof. James Lu, Prof. Jun Luo, Prof. Ken Mandelberg, Prof. James Taylor, and Prof. Alessandro Veneziani for helping me in study and research.

To my family

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Modern location-based services are collecting and processing spatio-temporal information of huge amount of continuous moving objects. Database systems supporting massive real time updates and efficient predictive queries [41, 49] over the moving objects have been extensively studied and are becoming increasingly important for the emerging location aware applications including real-time ride sharing (e.g. Uber) and location based crowd sourcing (e.g. Waze). One of the most extensively studied type of predictive queries is the *range query* (e.g. [21, 48, 59]). A predictive range query retrieves the objects locating within a spatial query region R at a future query prediction time t. Another popular type of query in location-based services is the *nearest neighbors query* (e.g. [3, 37, 59]). A predictive k nearest neighbor (kNN) query with a query point P retrieves the k objects that no other objects are nearer to P at the query prediction time t.

The naive way of processing theses queries is to linearly scan all objects and then check their validities regarding to the query predicates, which is slow when the number of moving objects is huge. A typical strategy for efficiently

answering a predictive query is first filtering out the objects that are likely not in the query result via a pruning mechanism and then for each remaining object, verifying its validity with the *query predicate* using a certain *prediction function* [18,41,48,49]. We call objects surviving the pruning step the *candidate objects* and the set of candidate objects the *candidate set*. The pruning step is critical for efficiency purposes since the major computational burden resides in the verification step. i.e. the more objects pruned, the less the query processing time.

Spatial accessing or indexing mechanisms are usually applied on moving object databases (MODs) to perform the pruning operation. Basically, these indexing mechanisms can filtered out un-qualifying index nodes or grid cells for predictive range searches based on objects' *motion functions* [41, 48, 49]. Then validities of the remaining objects regarding to the query predicate are computed using their location and motion information. Existing indexes for MODs can be categorized into tree-based indexes (e.g. [7, 8, 18, 19, 41, 47, 49, 57, 61]) and grid-based indexes (e.g. [36,44–46]) based on the underlying data structure storing the moving object information. Since moving objects like vehicles in the road networks have to frequently update their information in order to keep the system up-to-date, the information update rates are usually very high in real world location aware applications. Thus not only query performance but also update overhead must be considered while indexing MODs.

I summarize a few challenges and limitations of existing indexing structures for moving objects as below.

- First, tree-based indexes typically index object locations with tree structures like R-trees [14]. Consider that, real world moving objects like vehicles have a variety of different velocities, partitioning the moving

objects according to their velocities can reduce *search space expansion* of the index tree, which measures the expanding area of all index nodes over time, and thus reduce the query processing time. Although some existing works proposed velocity-based partitioning techniques on tree-based indexes, they rely on some heuristics and did not provide an optimal partitioning strategy.

- Second, although sophisticated maintenance mechanisms that optimize disk I/Os make tree-based indexes considerably efficient in in-disk scenarios, however, they are inefficient in in-memory scenarios, especially when the update rates are high. Simple (uniform) grids that require minimal maintenance efforts, have been proved to be more efficient than most tree-based structures for in-memory indexing. However, existing grid-based indexes do not consider velocity information, which can significantly affect the query performance. Moreover, the update operation for existing grid-based indexes is also not optimized.

- Finally, existing approaches for processing predictive range queries use motion function based pruning strategies with tree-based or grid-based indexes, which is not effective for long-term predictive range queries, since object velocities or motions can change over time and motion functions cannot effectively capture the objects' transition patterns far into the future. Generally speaking, existing pruning strategies for long-term predictive range queries are based on transition models learned from objects' historical trajectories. However, existing pruning approaches for long-term predictive range queries suffer from limited effectiveness due to the limited representation capacities of the underlying transition models.

To address these challenges, we propose a few enhanced indexing structures and pruning mechanisms for long-term predictive range queries.

- First, we proposed an optimal partitioning technique over speed values and directions of the moving objects for tree-based indexes based on sophisticated analysis on the search space expansion. Our speed partitioning technique is generic and can be used to improve query performances for various tree-based indexing structures.

- Second, we proposed the D-Grid, an in-memory uniform grid index in the velocity-location dual space, which augments tradition (location) grid-based indexes with a grid in the velocity domain. D-Grid significantly outperforms existing uniform grid indexes in terms of query performance. We also propose a *lazy deletion and garbage cleaning* mechanism that reduces update costs for D-Grid as well as other uniform grid indexes.

- Finally, we proposed a (high-order) Markov chain based method that efficiently and effectively reduces the candidate set for long-term predictive range queries, thus significantly reduces the query processing time. The key to our approach is to devise compressed representations for sparse multi-dimensional matrices, and leverage efficient algorithms for matrix computations.

1.1 Velocity-Based Partitioning for Tree-Based Indexes

In most real world applications, moving objects usually exhibit particular patterns on velocities (including speed values and directions). Therefore, velocity-

based partitioning can be applied to the indexes to improve performances of the indexes. Zhang et.al. [58] proposed the first idea of velocity-based partitioning for indexing moving objects. In their method, they first find k velocity *seeds* which maximize the *velocity minimum bounding rectangle* (VMBR), then partition the moving objects by assigning them to the nearest seed. In this way, the moving objects are partitioned into k parts and the VMBR for each part is minimized. Nguyen et.al. [33] proposed another velocity-based partitioning technique that partitions the indexes based on directions of the moving objects. This method partitions the moving objects based on their distance to the so-called *dominant velocity axes* (DVAs) in the velocity domain.

Speed values of the moving objects are always characterized by both the nature of the moving objects and the environment. For example, walking speeds for human beings range from 0 mph to 4 mph; driving speeds for vehicles in city road networks range from 0 mph to 100 mph; cruising speeds for airplanes usually range from 500 mph to 600 mph. Moreover, in most city road networks, speed values of the vehicles are also characterized by the categories of the roads. For example, most vehicles drive between 50-80 mph on highways, and 20-40 mph on street ways or even slower when the roads are busy. Such distributions of speed values of the moving objects can have significant impacts on query performances of the indexes. Query performances of typical tree-based indexes for MODs can be estimated by the average number of node accesses [49]. However, high speed moving objects will significantly enlarge the spatial areas of the index nodes containing them, which will likely incur unnecessary accesses to the low speed ones within the same nodes while processing queries. Thus partitioning the indexes by speed values of the moving objects can significantly improve query performance. Moreover, partitioning will reduce the number of

objects in each index partition, which also helps accelerate update operations.

Motivated by above observations, we proposed the novel speed partitioning technique. The proposed method first computes the optimal points (ranges) for speed partitioning. Then an optional second-level partitioning, based on directions of the moving objects, is performed within each speed partition. Note that the distributions of locations and speeds might change as time elapses that leads to changes of the optimal speed partitioning. Our proposed system can handle these changes through periodical partition update routines. Moreover, the speed partitioning technique is generic and can be applied with various tree-based indexes. The query performance of grid-based indexes depend on different factors, as the grid cells might contain quite different number of objects. We explore velocity information with enhanced grid-based indexes in the our next work.

1.2 Indexing with Uniform Grids

Earlier, most of the indexes resided in disks that have large storage capacity and are economical. Tree based methods (e.g. [18,41,47,49]) are popular structures for in-disk indexes, due to sophisticated maintenance mechanisms that optimize disk I/O. However, developments in computer hardware have made it feasible and affordable to process data on millions of moving objects in main memory. Moreover, in most moving object applications, such as traffic monitoring, crowd tracking, and games, update rates are extremely high, which requires the indexes to support both queries and updates efficiently. Therefore, in-memory indexing for moving objects is rapidly gaining popularity in recent research [44–46, 52, 53].

Although many efforts have been devoted towards reducing the burden of updates for tree based indexes [5, 25, 32, 47, 61] and extend them to in-memory environments [16,22,38,44,52], recent works [44,45] imply that, with in-memory settings, simple uniform grids are usually more efficient than tree based structures. The reason is that disk I/Os are eliminated for in-memory indexes, consequently, the sophisticated designs that aim to reduce disk I/O costs become cumbersome since they make update operations complicated and time consuming. On the contrary, simple uniform grids require minimal maintenance efforts, and thus are appropriate structures for in-memory indexes. The u-Grid [44] is one of the first uniform grid indexing structure in main memory for moving object databases. u-Grid achieves impressively high performance for both update and query processing, benefiting from a lightweight but efficient grid structure.

u-Grid uses only location information of the moving objects but disregards the velocity information, which we believe can further improve query performance. In this paper, we proposed a D-Grid (short for dual space grid) structure, which augments u-Grid and indexes moving objects in the (location-velocity) dual space. The dual space is a $2d$-dimensional Euclidean space, with the first d dimensions for velocity and the other d dimensions for location. Similar to u-Grid, our proposed D-Grid is again based on a uniform grid structure. We also propose a *lazy deletion and garbage cleaning* (LDGC) mechanism that can be applied on generic grid based indexing structures and further improves update performance for moving object databases. Although similar *lazy update* mechanisms have been proposed for tree based indexes (e.g. [24, 47, 51, 61]), to the best of our knowledge, no existing work extends this technique to grid based indexes. Through the LDGC mechanism, the object data to be deleted

is marked as invalid but not immediately erased from the index. When the number of invalid entries in the grid cell exceeds a predefined threshold, the garbage cleaning operation is invoked to clear the corresponding invalid entries. The LDGC mechanism significantly reduces the amortized computation costs for *non-local updates* (see Chapter 4). Based on the D-Grid structure, we propose algorithms for both predictive range queries and predictive k nearest neighbor queries. D-Grid outperforms existing uniform grid indexes in terms of query performance since it uses velocity information of the moving objects to reduce the search space of range searches and consequently reduces the total number of objects retrieved for answering the queries. In addition, D-Grid is easy to parallelize similar to other uniform grid structures [45, 46]. We only considered single threaded environments in this work and leave multi-threaded versions of D-Grid to future work.

1.3 Processing Predictive Range Queries

The typical approach for answering a predictive range query is first filtering out the objects that are likely not in the query result via a pruning mechanism and then for each remaining object, verifying its validity with the *query predicate* using a certain *prediction function* [18, 41, 48, 49]. We call objects surviving the pruning step the *candidate objects* and the set of candidate objects the *candidate set*. The pruning step is critical for efficiency purposes since the major computational burden resides in the verification step. i.e. the more objects pruned, the less the query processing time.

For short-term predictions (in the order of tens of seconds), motion functions, including linear motion functions [41, 49] and recursive motion func-

tions [48], are effective and efficient for answering predictive range queries via various indexing techniques that usually augment R-trees [14] or grids. R-tree based indexes use *minimum bounding rectangles* (MBRs) that evolve with the underlying motion functions to group the moving object locations over time. Objects in the MBRs that do not intersect with the range query window at the prediction time are pruned. However, R-tree based indexing mechanisms suffer from high update burden, despite many efforts (e.g. [24, 47, 51]) devoted towards reducing update costs. A uniform grid, that partitions the predefined space domain into grid cells with equal and fixed length, has recently been proven efficient in terms of both queries and updates in main memory [44, 45]. Grid based indexing methods use the *query window enlargement* technique [18] for pruning and processing predictive range queries. Specifically, objects in the cells that do not intersect with the enlarged query window are pruned. After pruning, the underlying motion functions are used to estimate the object locations at the prediction time for verification.

Unfortunately, motion functions cannot accurately perform long-term predictions (in the order of tens of minutes), since the motions of moving objects can change over time. Trajectory pattern based methods (e.g. [20, 28–30, 40, 54–56]) and descriptive model based methods (e.g. [10, 13, 23, 26, 60]), which learn from historical trajectories, can be used, instead, for long-term predictions. However, trajectory pattern and descriptive model based predictions are usually much more expensive, thus effective pruning is even more critical in these scenarios. Motion function based pruning strategies [18, 41] as we discussed earlier are not effective for long-term predictive range queries since they do not capture the motion changes over time. Other pruning methods, such as the *travel time grid* that records the average travel time between grid cells

in the space domain [15], rely on simple predicates and have limited pruning capacities.

We proposed a (high-order) Markov chain based method for efficiently and effectively pruning the search space for long-term predictive range queries. Specifically, we partition the predefined space domain with a uniform grid, where each grid cell is called a *state*. A *path* of a moving object is defined as a sequence of states on the grid. We assume that the paths follow an order-k Markov chain model, where the *transition matrix*, describing *transition probabilities* between states, is off-line learned from historical trajectories. Given a predictive range query, through certain matrix computations, we prune the paths that are not promising to transit into the query window at the prediction time and then use any state-of-the-art prediction method for verification. Computation involving the Markov transition matrix can be very expensive in terms of both CPU and memory. Based on the observation that, in real world scenarios, the Markov transition matrices are extremely sparse, especially those for high-order Markov chains, we proposed a novel approach to compactly store the sparse transition matrices in main memory, novel algorithms for performing the arithmetic operations involved in our pruning mechanism.

1.4 Contribution

The contributions of this dissertation are summarized as follows.

- The optimal speed partitioning technique.
  - We proposed a novel method for estimating the search space expansion which can be used as a generic cost metric to estimate query performance of tree-based indexes for MODs.

  - We proposed the novel optimal speed partitioning technique which minimizes search space expansion of the indexes using dynamic programming.

- The dual space grid indexing structure.

  - We proposed D-Grid, the first dual space uniform grid index in main memory for moving object databases, which provides performance improvements in query processing by almost an order of magnitude.
  - We proposed the lazy deletion and garbage cleaning mechanism that can be applied to both D-Grid and existing uniform grid indexes for accelerated update processing.

- Markov chain based pruning approach for long-term predictive range query.

  - We proposed an effective and efficient pruning algorithm, based on (high-order) Markov chain models, to reduce the search space for predictive range queries.
  - We proposed a novel approach to compactly store sparse multi-dimensional matrices and efficiently support the arithmetic operations involved in our pruning mechanism.

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter 2 reviews the related work on indexing moving objects and processing predictive spatio-temporal queries. Chapter 3 discuss velocity-based partitioning techniques

on tree-based indexes of MODs for improving query performance. Chapter 4 discuss a uniform dual-space grid structure that indexes moving objects in main memory and provide both efficient update and query operations. Chapter 5 discuss a (high-order) Markov chain based pruning mechanism for processing long-term predictive range queries. Chapter 6 concludes this dissertation and gives future works.

Chapter 2

Related Work

In this chapter, I review related works on indexing moving objects and processing predictive range queries.

2.1 Tree-Based Indexes for MODs

Saltenis et al. [41] proposed the TPR-tree (short for Time-Parameterized R-tree) that augments the R*-tree [2] (a variant of the R-tree [14]), with velocities to index moving objects with motion functions. Specifically, an object in the TPR-tree is indexed by its time-parametrized position with respect to its velocity vector. A node in the TPR-tree is represented by a *minimum bounding rectangle* (MBR) and the velocity on each side of the MBR which bounds all moving objects contained in the corresponding MBR at any time in the future. The TPR-tree uses time-parameterized metrics when choosing the target nodes for insertion and deletion. The time-parameterized metric is calculated as $\int_{t_l}^{t_l+H} A(t)dt$, where $A(t)$ is the metric used in the original R-trees. H is the *horizon* (the lifetime of the node) and t_l is the time of an insertion or the index creation time. The TPR-tree uses a step-wise greedy strategy to

choose the MBR where a new object is inserted. Since the objects are moving as time passes, the overlaps between MBRs become larger, which eventually makes the step-wise greedy strategy ineffective. Tao et al. proposed the TPR*-tree [49] that uses the same data structure as the TPR-tree with optimized insertion and deletion operations, which significantly reduce the overlaps between MBRs. Unfortunately, TPR-trees and TPR*-trees are inefficient for updates due to their sophisticated node splitting and data reinsertion operations. Many efforts have been devoted toward reducing the burden of updates in R-tree based indexes. Kwon et al. proposed the LUR-tree [24] that updates the R-tree structure only when an object leaves its MBR. Xiong et al. proposed the Rum-tree [47,51] that performs lazy deletions on the R-tree through the so called *update memo*, which keeps multiple versions of the same object. Zhu et al. recently proposed the Rum$^+$-tree [61] that is an extension of the Rum-tree and further improves the update performance with a hash table.

Besides R-trees, B$^+$-trees and quadtrees [11] can also be used to index moving objects. The B^x-tree, proposed by Jensen et al. [18], is the first indexing approach based on B$^+$-tree. The B^x-tree uses space-filling curves, such as Z-curves and Hilbert curves, to map the d-dimensional locations into scalars that can be indexed by B$^+$-trees. The time axis is partitioned into intervals of duration Δt_{mu}, which is the maximum duration in-between two updates of any object location. Each such interval is further partitioned into n equal-length *phases* and each phase is associated with a *label timestamp*. Instead of indexing the object locations at their update timestamps, the B^x-tree indexes the locations at the nearest future label timestamp. After each $\Delta t_{mu}/n$ timestamps, one phase expires and another is generated. This rotation mechanism is essential to preserve the location proximity of the objects. The B^{dual}-tree [57] and

STRIPES [36] index the moving objects in $2d$-dimensional *dual space*, which combines the d dimensional velocity space and the d-dimensional location space. The B^{dual}-tree uses a $2d$-dimensional Hilbert curve to map the underlying dual space coordinates to scalars and then indexes the scalars with B^+-trees, while STRIPES indexes the dual space using a regular hierarchical grid decomposition indexing structure, which is essentially an in-disk PR quadtree (P for point and R for region) [42].

Velocity-based partitioning techniques, which utilize the velocity information from a global perspective, are used to further improve the query performance of MOD indexes. Intuitively, velocity-based partitioning can improve query performance because search space expansion (defined as the enlargement of the index nodes) [33] of the partitioned indexes considerably decreases in some scenarios. Zhang et.al. [58] firstly defined the VMBRs which represent the minimal rectangles in the velocity domain that bound the velocity vectors of all moving objects and proposed the partitioning method that minimizes the VMBRs within each partition. At the first step of this method, given the number of partitions k, the velocity vectors of exactly k moving objects that form largest VMBR are selected as seeds for the k partitions. Then each object is assigned to the partition with minimum VMBR increase. This method has some limitations. Firstly, it is difficult to determine the number of partitions k. Secondly, the partitioning might be far from optimum since this method relies on very simple heuristics and does not perform any analysis on search space expansion. Thi et al. [33] proposed the partitioning technique based on DVAs in the velocity domain. They applied principal component analysis and K-means clustering on the velocities of the moving objects to find k-1 DVAs. Then the velocity domain is partitioned into k partitions according to the DVAs, one

partition for each DVA plus one outlier partition. Each moving object is assigned to the nearest DVA partition if the distance between its velocity vector and the DVA is smaller than a threshold, otherwise it will be assigned to the outlier partition. Through this partitioning method, the velocity domain is reduced to nearly 1-dimensional parts, which dramatically reduces the search space expansion. However, this method still requires the number of partitions k as a parameter. Moreover, the performance of this method will significantly reduce if the velocity domain has no effective DVAs.

2.2 Grid-Based Indexes for MODs

In modern computing machines, large amounts of moving object data can be stored in main memory, making in-memory indexes for moving object databases effective. Although (tree based) in-disk indexes can be directly transfered to main memory [8,44,52], recent research (e.g. [38,44,45]) indicates that a simple, uniform grid might be the best choice for managing moving objects in main memory. Sidlauskas et. al. proposed the u-Grid [44] coupling a uniform grid as the primary index and a secondary index on object IDs. Since no grid refinement or re-balancing needs to be performed during updates, uniform grids require less maintenance efforts than adaptive grids, such as the grid file [34] and hierarchical space partitioning methods, such as quadtrees [11]. Figure 2.1 illustrates the structure of u-Grid. The uniform grid covers a predefined location space, stored as a 2-dimensional array, where each element of the array corresponds to a grid cell with fixed side length. Each grid cell links to a list of buckets that store the object information, including object ID and the spatial coordinates. u-Grid uses buckets instead of a simple linked list since the data is

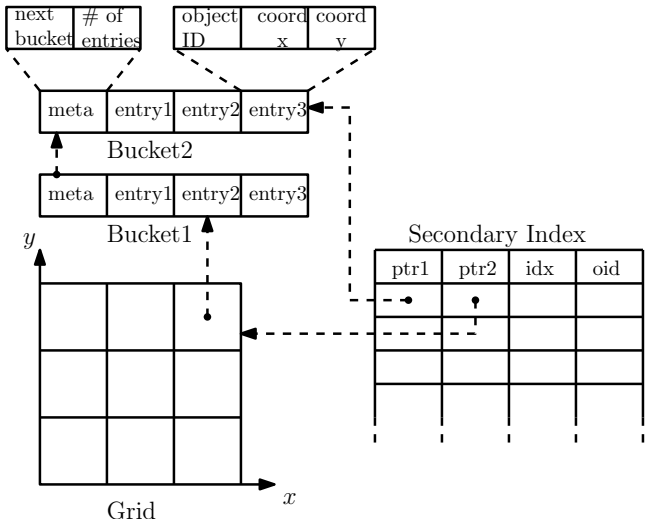


Figure 2.1: Structure of u-Grid

loaded in blocks (cache lines) to the CPU cache and large buckets increase data access locality thus provide more efficient data fetching [9]. Each bucket also has a meta-data field containing a pointer to the next bucket and the current number of entries (objects) in the bucket. To avoid expensive index searches for updates, u-Grid employs a secondary index that uses object ID (*oid*) as the index key and links to the corresponding data entry in the primary index (u-Grid). Thus the secondary index provides direct access to the object data in the primary index. As shown in Figure 2.1, in the secondary index, $ptr1$ points to the bucket where the object data is stored, $ptr2$ points to the corresponding grid cell, and the *idx* field stores the in-bucket position of the object data.

Another advantage of uniform grid indexes concerns the ease of parallelism. Sidlauskas et. al. proposed the TwinGrid [43] and the PGrid [45] which augment u-Grid and provide highly parallel update and query algorithms. PGrid solves problems of stale query results and wasted CPU cycles that exist in TwinGrid by careful use of light-weight locking techniques. Ray et al. recently proposed PASTIS [38] which decomposes the spatial domain into grid cells and,

for each grid cell, maintains a partial temporal index for moving objects that visited the cell. Updates for different grid cells are concurrently processed in separate threads.

2.3 Processing Predictive Range Queries

R-tree based indexes use *minimum bounding rectangles* (MBRs) that evolve with the underlying motion functions to group the moving object locations over time and also MBRs to prune the search space for predictive range queries. Specifically, a node is pruned if its MBR does not intersect with the range query window at the prediction time. On the other hand, uniform grid based indexing mechanisms apply the *query window enlargement* technique, proposed in [18], instead of MBR enlargement to prune the search space for predictive range queries. Grid cells that do not intersect with the enlarged query window are pruned for the query. The query windows are enlarged with the *enlargement speeds* on each side. Generating the enlargement speeds is accomplished in two steps. Firstly, preliminary enlargement speeds are set as the maximum speeds of all objects. Then, final enlargement speeds are computed with the aid of the *velocity histogram*, which is a 2-dimensional grid that captures the maximum and minimum projections of velocities onto each axis of the objects in each cell.

Unfortunately, enlarging MBRs or query windows cannot monitor positions of the objects far into the future, as their motions might change over time. Hendawi et. al proposed the long-term predictive query processor named *Panda* [15]. The pruning strategy employed by Panda is based on the *travel time grid* (TTG), which is a 2-dimensional array where each cell $TTG[i, j]$ stores average travel time between two grid cells C_i and C_j that are learned

from historical trajectories. According to TTG, objects in the cells that are unreachable to the query window within the prediction time are pruned. However, the average travel time is far too simple a measure to capture the transition patterns between these cells. For example, in real world applications, where travel time are affected by traffic conditions, the average travel times might be useless for real-time travel time estimation. Emrich et.al [10] proposed a framework modeling and querying probabilistic spatio-temporal data. Specifically, this method models possible object trajectories by time-homogeneous order-1 Markov chains and computes, for each state s, the probability that s transits into the query window within a specified time range. However, we will discuss later in Chapter 5 that low order Markov chains are not suitable for trajectory prediction but are appropriate for pruning, i.e. objects starting from s can be pruned if the probability that s transits into the query window at the prediction time is below a predefined threshold.

Chapter 3

The Speed Partitioning Technique

In this chapter, I present a novel speed partitioning technique based on a formal analysis over speed values of the moving objects. We first formulate the optimal speed partitioning problem based on search space expansion analysis and then compute the optimal solution using dynamic programming. We then build the partitioned indexing system where queries are duplicated and processed in each index partition. Extensive experiments demonstrate that our method dramatically improves the performance of indexes for moving objects and outperforms other state-of-the-art velocity-based partitioning approaches.

3.1 The Optimization Speed Partitioning Problem

In this section, we introduce the notion of search space expansion which can be used as a generic cost metric to estimate query performance of tree-based

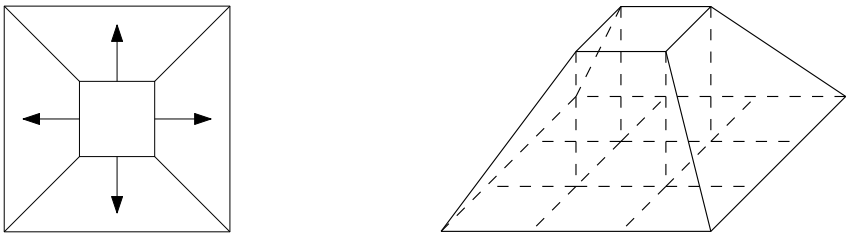
indexes for MODs. We then present the method for computing search space expansion and formulate the optimal speed partitioning problem.

3.1.1 Search space expansion

Figure 3.1(a) shows a typical example of how the geometry area of an index node expands. In this figure, the moving objects are originally located in a square area (the inner one) and move in arbitrary directions. At some future time, the objects will spread in a larger square area (the outer one). We model the expansion of the node as a trapezoid prism where the top base is the original area and the bottom base is the future area of the node. Figure 3.1(b) illustrates such a trapezoid prism of the node in Figure 3.1(a). The volume of the trapezoid prism corresponding to an index node is called the search space expansion of this node. The sum of search space expansions of all index nodes is called the search space expansion of the index. A formal definition of search space expansion is given in Definition 3.1.

Definition 3.1. Search space expansion. Given any node in an MOD index I, its area at time t is $S(t)$. The search space expansion of the node from time 0 to any future time t_h is $\nu(t_h) = \int_0^{t_h} S(t)dt$. The search space expansion of the index is the sum of the search space expansions of all nodes: $V(t_h) = \sum_{\forall node \in I} \nu(t_h)$

If queries are randomly generated in the predefined space domain, nodes with larger search space expansions have higher probabilities to be accessed to answer the queries [49]. Consequently, indexes with smaller search space expansion enjoy better query performance. Thus we wish to find a partitioning strategy that minimizes the search space expansion of the indexes, i.e. the volumes of all trapezoidal prisms, in order to minimize query costs.



(a) Index node (b) Search space expansion

Figure 3.1: Search space expansion of an index node

We propose the speed partitioning technique which partitions the indexes based on speed values of the moving objects. Since the moving objects are separated based on their speed values, thus fast growing nodes for high speed objects will not affect those for low speed objects. Therefore the search space expansion of an index will be dramatically reduced if we conduct appropriate partitioning on speed values. In the next subsection, we will discuss how to achieve the optimal index partitioning based on speed values. Note that in our analysis, we only consider the search space expansions of leaf nodes, because in most scenarios the number of leaf nodes significantly exceeds that of internal nodes.

3.1.2 The optimal speed partitioning

Our speed partitioning technique is based on solving the optimal speed partitioning problem, thus is different from and more generic than all state-of-the-art velocity-based partitioning techniques [33,58] that rely on some kinds of heuristics. We now formalize the optimal speed partitioning problem that minimizes search space expansion.

Denote $\mathcal{O} = \{o_1, o_2, \cdots, o_N\}$ as the set of moving objects and denote the speed of object o_l as v_{o_l}. Let $\Omega = \{v_1, v_2, \ldots, v_q\}$ represent the speed domain,

where $v_1 < v_2 < \cdots < v_q$. Thus for all $o_l \in \mathcal{O}$, we have $v_{o_l} \in \Omega$. We note that in most applications the speed domain can be easily discretized into finite number of different speed values. Let $v_0 = v_1 - \epsilon$, where ϵ is a positive number and $\epsilon \to 0$. v_0 is a dummy speed used for simplifying notations. Let $\Omega^+ = \Omega \bigcup \{v_0\}$.

Now let $\Delta = \{\delta_0, \delta_1, \cdots, \delta_k\}$, $1 \leqslant \delta_i \leqslant q$, where $\delta_0 = 0$ and $\delta_k = q$. Therefore Δ partitions the speed domain into k (non-overlapping) parts, denoted as $\Omega_i = (v_{\delta_{i-1}}, v_{\delta_i}]$, $1 \leqslant i \leqslant k$. We say Δ is a partitioning on Ω. Meanwhile, $\mathcal{O}$ is partitioned accordingly into k parts: $P_i (1 \leqslant i \leqslant k)$, where $P_i = \{o_l : v_{o_l} \in (v_{\delta_{i-1}}, v_{\delta_i}]\}$. We denote I_i as the corresponding indexing tree, such as the B^x-tree or the TPR*-tree, for P_i. Note that k is automatically computed rather than an input of our method.

Our goal is to find the optimal partitioning, denoted as $\Delta^\star$, that minimizes the overall search space expansion of all index partitions. We can achieve this goal by solving the following minimization problem:

$$\Delta^\star = \arg\min_{\Delta} \{v_{\delta_0} < v_{\delta_1} < \cdots < v_{\delta_k} : V(t_h)\} \tag{3.1}$$

where $V(t_h) = \sum_{0<i\leqslant k} V_i(t_h)$ represents the overall search space expansion of all index partitions and $V_i(t_h)$ the search space expansion of partition I_i. t_h is the maximum predict time for the predictive queries [41, 49]. Without loss of generality, we present next how to compute $V_i(t_h)$.

According to Definition 3.1, in order to compute $V_i(t_h)$, we first need to compute the search space expansion of every single index node in I_i which requires 1) the initial node area, and 2) the expanding speed of each node. We present the approach to compute $V_i(t_h)$ step by step in the following paragraphs.

Generate uniform regions In most real world applications, the moving objects may not be uniformly distributed. Thus before calculating the search space expansion, we first divide the space domain into subregions such that the moving objects in P_i are (close to) uniformly distributed within each subregion. Uniformity will not only significantly reduce the complexity of calculation but also help obtain more accurate estimations. We will introduce a quad tree based method to find the uniform subregions in Section 3.2. We denote the set of uniform subregions of P_i as $\mathcal{R}_i = \{R_{i1}, R_{i2}, \ldots, R_{im_i}\}$.

Compute initial node area Now we compute the initial areas of the nodes within subregion R_{ij}, where $0 < j \leqslant m_i$. Without loss of generality, we assume R_{ij} to be a square area with side length of D_{ij}. We also consider the index nodes as square shaped with expected side length of d_{ij} and let c represent the expected number of objects in each node. c is determined by the storage size of each node which is a parameter in our method. Since moving objects are uniformly distributed in R_{ij}, we have $\frac{d_{ij}^2}{c} \propto \frac{D_{ij}^2}{N_{ij}}$ where N_{ij} represents the number of objects in R_{ij}. Thus d_{ij} can be estimated as $d_{ij} = D_{ij}\sqrt{\frac{c}{N_{ij}}}$.

Compute expanding speed Next we introduce the method for estimating expanding speeds of the index nodes in R_{ij}. Since we make no assumptions on the patterns of the moving objects' directions, we consider that the objects in each node travel at arbitrary directions. Thus every single node expands with equal speed in all directions while the expanding speed is the maximum speed value of the moving objects in the corresponding node.

Let H_{iju} represent the number of moving objects in R_{ij} whose speed values

fall in the range $(v_{\delta_{i-1}}, v_u]$, where $v_u \in \Omega$ and $\delta_{i-1} < u \leqslant \delta_i$, formally

$$H_{iju} = |o_l \in R_{ij} : v_{\delta i-1} < v_{o_l} \leqslant v_u| \tag{3.2}$$

Since the speed values of the moving objects are independent given a certain speed distribution, expanding speed of any node in R_{ij} is v_u with the probability

$$p(i,j,u) = \frac{\binom{H_{iju} - H_{ij\delta_{i-1}}}{c} - \binom{H_{ij(u-1)} - H_{ij\delta_{i-1}}}{c}}{\binom{N_{ij}}{c}} \tag{3.3}$$

where $\binom{a}{b}$ is a combination number.

Compute search space expansion For each speed partition, we can apply a second-level (direction-based) partitioning into 4 quadrants as illustrated in Figure 3.2 if it further improves search space expansion. Hence we compute the search space expansion of R_{ij} both with and without the second-level partitioning and select whichever achieves smaller value. When no second-level partitioning is performed, the search space expansion of a single node in R_{ij} can be calculated by

$$\nu^1(t_h) = \nu(t_h, v_u) = \int_0^{t_h} (d_{ij} + 2v_u t)^2 \, dt \tag{3.4}$$

When the second-level partitioning is further applied, we compute the search space expansion for each quadrant. Expected side length of the nodes in the quadrant partitions is $2d$ and the search space expansion is calculated by

$$\nu^2(t_h) = \nu(t_h, v_u) = \int_0^{t_h} (2d_{ij} + v_u t)^2 \, dt \tag{3.5}$$

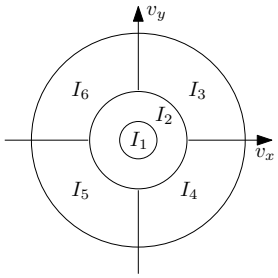


Figure 3.2: Speed partitioning

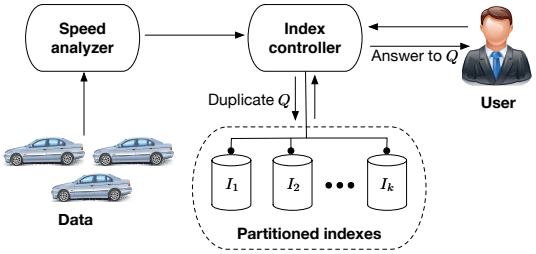


Figure 3.3: System architecture of SP

Therefore, the expected search space expansion of all nodes in R_{ij} can be calculated by

$$V_{ij}(t_h) = \left\lceil \frac{N_{ij}}{c} \right\rceil \sum_{v_u \in \Omega, \delta_{i-1} < u \leqslant \delta_i} \nu(t_h) p(i,j,u) \tag{3.6}$$

where $\left\lceil \frac{N_{ij}}{c} \right\rceil$ computes the total number of nodes in R_{ij} and $\nu(t_h)$ represents the minimum of $\nu^1(t_h)$ and $\nu^2(t_h)$. Finally, the overall search space expansion $V(t_h)$ is calculated by

$$V(t_h) = \sum_{1 \leqslant j \leqslant k} \sum_{0 < j \leqslant m_i} V_{ij}(t_h) \tag{3.7}$$

3.2 The Partitioned Indexing System

Based on the above analysis on search space expansion, we propose the speed partitioning technique (SP) for indexing moving objects. Figure 3.3 illustrates the system architecture of SP. SP uses a centralized indexing system consisting of three parts: the speed analyzer, the index controller, and the partitioned indexes. The speed analyzer receives data from the moving objects and computes the optimal speed partitioning. The index controller then creates the corresponding partitioned indexes. Once receiving queries from users, the in-

dex controller duplicates the queries and push them to the index partitions. After all index partitions finish processing the queries, the index controller collects and integrates the query results and sends them back to users. We will discuss more details of SP in the remainder of this section.

3.2.1 The optimal speed partitioning

In this subsection, we discuss how to find the optimal speed partitioning through dynamic programming. Let $\Lambda_r^\star$, $0 < r \leqslant q$, be a sequence $(\lambda_0, \lambda_1, \cdots, \lambda_r)$ where $v_{\lambda_i} \in \Omega^\star$ and $0 = \lambda_0 < \lambda_1 \leqslant \cdots \leqslant \lambda_{r-1} \leqslant \lambda_r = r$. The set of distinct values in $\Lambda_r^\star$ form the optimal partitioning of the sub speed domain of $(v_0, v_r]$, denoted as $\Delta_r^\star$. Thus our goal is to find $\Delta_q^\star$.

In order to compute $\Delta_q^\star$ using dynamic programming, we need to maintain two arrays $\mathbb{V}^\star$ and $\mathbb{T}^\star$, where $V_r^\star$ and $T_r^\star$ (the r^{th} values of $\mathbb{V}^\star$ and $\mathbb{T}^\star$) store the search space expansion of $\Delta_r^\star$ and the r^{th} value ($\lambda_{r-1}^\star$) in $\Lambda_r^\star$, respectively. $V_r^\star$ and $T_r^\star$ can be computed by Equation (3.8) and (3.9), respectively.

$$V_r^\star = \begin{cases} 0 & r = 0 \\ \min\limits_{0 \leqslant s < r} \{V_s^\star + V_{(v_s, v_r]}\} & 0 < r \leqslant q \end{cases} \tag{3.8}$$

$$T_r^\star = \arg\min_{0 \leqslant s < r} \{V_s^\star + V_{(v_s, v_r]}\}, 0 < r \leqslant q \tag{3.9}$$

where $V_{(v_s,v_r]}$ is the search space expansion of partition $P_{(v_s,v_r]}$ and $P_{(v_s,v_r]} = \{o_l : v_{o_l} \in (v_s, v_r]\}$. Note that we define $V_0^\star = 0$ in order to simplify denotations. Next we discuss how to compute $V_{(v_s,v_r]}$, for all $(v_s, v_r] \subset \Omega$.

In order to compute $V_{(v_s,v_r]}$ using Equation (3.7), we first need to generate the uniform subregions mentioned in Section 3.1. We propose a quad tree [11] based method to generate the uniform subregions for every $P_{(v_s,v_r]}$. We

Algorithm 3.1: Generate uniform subregions

```
   input  : 𝒬: a set of quad tree nodes
   output: ℛ: a set of uniform subregions
   /* check the uniformity of the current nodes                    */
1  if ∀Q_j ∈ 𝒬, Q_j is uniform then
2  |  add the region of 𝒬 to ℛ;
3  else
   |  /* explore the child nodes                                   */
4  |  for i ← 0 to 3 do
5  |  |  foreach Q_j ∈ 𝒬 do
6  |  |  |  CQ_j ← Q_j.child[i];
7  |  |  generate uniform subregions on 𝒞𝒬;
```

first divide the objects into q layers, where moving objects within the same layer have same speed values (represented by the average speed value in each layer). Each layer is divided into square subregions using a quad tree such that the objects in each subregion are uniformly distributed. We use χ^2-test (significance level 5%) to test the uniformity of each subregion. We also fix 5 as the maximum depth of the quad trees. In order to generate the uniform subregions for $P_{(v_s,v_r]}$, we need to combine the corresponding layers, layer $s+1$ through r. We choose the most fine grained division when the divisions of different layers conflict, thus objects in the subregions of the combined layer always contain uniformly distributed objects. Figure 3.4(left) shows an example of such layers, where there are 3 different speed values v_1, v_2, v_3 and the objects in the 3 layers are represented as squares, diamonds, and dots, respectively. Figure 3.4(right) shows the result of the merge operation.

Algorithm 3.1 shows the pseudo code for the merge operation. This is a recursive algorithm which takes a set of $r - s$ quad tree nodes (one node for each layer) as input. If objects within all the current nodes are uniformly distributed, we add the (square) spatial region represented by the quad tree

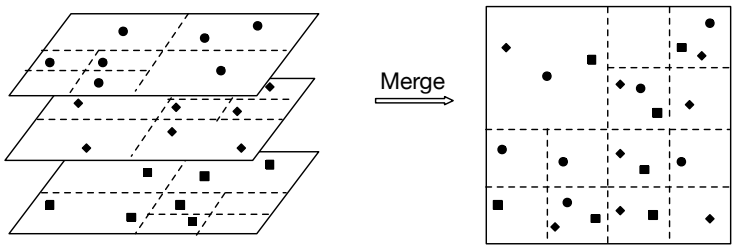


Figure 3.4: An example of merge

nodes into the result set (lines 1-2). Otherwise, we recursively explore the 4 child nodes (each 2-dimensional quad tree node has 4 child nodes) at the next level of the quad trees (lines 3-7). Note that the input nodes will always locate at the same positions in the corresponding quad trees for all recursive calls, since we set the root nodes of the quad trees as input of the initial call.

In order to find the optimal partitioning $\Delta_q^\star$, we need to compute $V_r^\star$ for each r $(0 < r \leqslant q)$. As shown in Equations (3.8) and (3.9), we iteratively find the best s which leads to the optimal partitioning on $(v_0, v_r]$ and stores it as $T_r^\star$. During the computation for $V_r^\star$, we can use previously computed optimal results on $(v_0, v_s]$, i.e. the values of $V_s^\star$ for each s $(0 \leqslant s < r)$. Finally, we can obtain the optimal partitioning on $(v_0, v_q]$ by tracking backwards the values in $\mathbb{T}^\star$, i.e. each $\lambda_i \in \Lambda_q^\star$ $(0 \leqslant i \leqslant q)$ can be computed by

$$\lambda_i = \begin{cases} 0 & i = 0 \\ T^\star_{\lambda_{i+1}} & 0 < i < q \\ q & i = q \end{cases} \tag{3.10}$$

Algorithm 3.2 shows the pseudo code of our dynamic programming based algorithm to solve the optimal speed partitioning problem. Algorithm 3.2 first creates the quad trees for uniform subregion generation (line 1). Then the search space expansions of partition $P_{(v_s,v_r]}$, for all $(v_s, v_r]$, are calculated (lines

Algorithm 3.2: Find the optimal speed partitioning

```
1  Create quad trees;
   /* Pre-compute search space expansion for partition P_(v_s,v_r]
      using Equation (3.7)                                        */
2  foreach (v_s, v_r] ∈ Ω do
3      V_(v_s,v_r] ← the search space expansion of P_(v_s,v_r];
   /* Iteratively compute V_r* and T_r* using Equation (3.8) and
      (3.9)                                                       */
4  V_0* ← 0;
5  for r ← 1 to q do
6      min ← inf;
7      for s ← 0 to r − 1 do
8          if V_s* + V_(v_s,v_r] < min then
9              min ← V_s* + V_(v_s,v_r];
10             T_r* ← s;
11     V_r* ← min;
   /* Compute the final results using Equation (3.10)             */
12 λ_q ← q;
13 for i ← q − 1 to 1 do
14     λ_i ← T*_(λ_{i+1});
15 λ_0 ← 0;
```

2-3). Then dynamic programming is used to compute the values of $V_r^\star$ and $T_r^\star$ based on Equations (3.8) and (3.9) (lines 4-11). Finally, λ_0 through λ_q are computed from $\mathbb{T}^\star$ using Equation (3.10) (lines 12-15). Note that we compute the search space expansion (line 3) both with and without the second-level partitioning as described in Section 3.1 and store the smaller value as $V_{(v_s,v_r]}$. The corresponding speed partition in the final result is further partitioned into four sub-partitions (one for each quadrant in the velocity domain) if it achieves smaller search space expansion. Figure 3.2 shows an example of the output of our algorithm. Actually, high speed partitions are more likely to be further partitioned into quadrants since direction has more impact on high speed partitions. The time complexity of Algorithm 3.2 is analyzed as follows.

Complexity analysis

Execution time of Algorithm 3.2 consists of three parts: 1) creating the quad trees takes $O(N)$ time; 2) pre-computing the search space expansions for each sub speed domain takes $O(q^2)$ time; and 3) the dynamic programming part also takes $O(q^2)$ time. Thus the total time complexity of Algorithm 3.2 is $O(N + q^2)$. Note that the analysis relies on the condition that maximum depth of the quad trees is fixed, as mentioned earlier in this section.

3.2.2 Index update

Index update of our system consists of two parts: object update and partition update. Object update corresponds to status (e.g. location and velocity) updates of the moving objects, which is essential to keep the objects' locations up-to-date. When a moving object updates its status, the index controller will determine whether it should be inserted into a different partition based on its current velocity. Then the object will be either deleted from its previous partition and inserted into the new one or simply updated in the previous partition. Note that each index partition contains only a portion of the moving objects, thus object update in the partitioned indexes takes less CPU time than that in the original index without partitioning.

Partition update corresponds to changes of the optimal speed partitioning. Since the objects are continuously moving, both their location and speed distributions might change over time. Thus we need to re-compute the uniform subregions as well as the optimal speed partitioning when necessary. We simply conduct partition updates periodically with cycle time customized according to the dataset. For example, in city road networks, location and speed distribu-

tions of the vehicles might be different between rush hours and regular hours, for which we can use hourly partition update routines.

3.2.3 Query processing

In this work, we consider predictive time-slice queries [41, 49] which retrieve tentative future locations of the moving objects. We evaluate both predictive range queries and predictive k nearest neighbor (kNN) queries in the experiments (Section 3.3). Specifically, a predictive range query is associated with two coordinates (bottom-left point and upper-right point of the range query window), while a predictive kNN query is associated with a coordinate (center of the kNN query) and kNN-k. Both of the two kinds of queries are associated with a query predict (future) time, which indicates that the queries are performed on the objects' predicted locations at that time.

Query processing for SP is straightforward. The original queries are duplicated (with modifications if necessary) and processed within each partition either concurrently or sequentially. In order to compare the performance between partitioned indexes and their unpartitioned counterparts, in this chapter, we conduct the duplicated queries sequentially. Within each index partition, queries are performed using the algorithm associated with the basic indexing structure (e.g. the B^x-tree or the TPR*-tree).

3.3 Experimental Study

In this section, we conduct extensive experiments to evaluate the performance of our speed partitioning technique with both main memory indexes and disk indexes. Both simulated traffic data and real world GPS tracking data are used

Table 3.1: Experimental settings

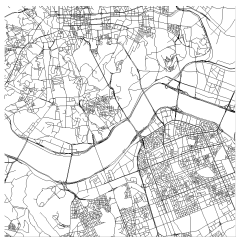
Parameter	**Setting**
Space domain(m×m)	**10,000×10,000**
Number of objects	**100K**, 200K, ..., 500K
Query window size (m×m)	200×200, **400×400**, ..., 1000×1000
kNN - k	10, 20, **30**, ..., 50
Query predict time (ts)	0, 30, **60**, ...,120
Node size (byte)	1K, 2K, **4K**,$\cdots$, 16K
datasets	SEO, **LD**, BOS, SZ

in the experiments. We evaluate both update throughput (average number of updates performed in a second) and query response time. Query response time consists of I/O latency and CPU time for disk indexes while only CPU time for main memory indexes.

We use the B^x-tree and the TPR*-tree as the basic indexing structures. We compare our approach of speed partitioning (SP-B^x and SP-TPR*) with the state-of-the-art approaches of DVA-based partitioning [33] (dVP-B^x and dVP-TPR*) and VMBR-based partitioning [58] (mVP-B^x and mVP-TPR*) as well as the baseline approaches (B^x and TPR*). We set the number of partitions k in DVA and VMBR-based partitioning techniques as 3 and 5, respectively, which is consistent with the experimental settings in the original papers. All algorithms are implemented with C++ language and all experiments are performed with 2.93GHz Intel Xeon CPU and 1TB RAM in CentOS Linux. The experimental settings are displayed in Table 3.1 where the default settings are boldfaced.

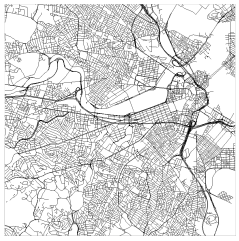
3.3.1 Dataset description

In this subsection, we introduce datasets used in the experiments. Figure 3.5 shows city road networks corresponding to the datasets in the experiments.



(a) Part of Seoul road network

(b) Part of London road network



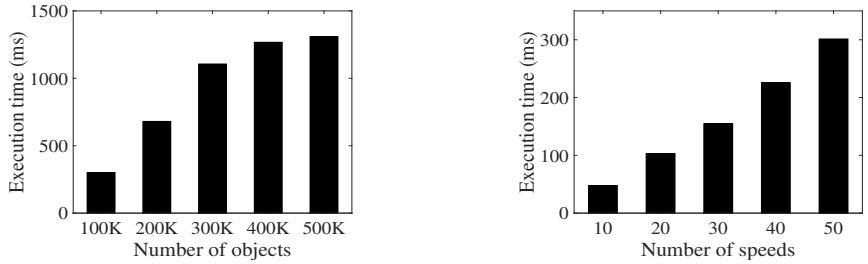
(c) Part of Boston road network

(d) Part of Shenzhen road network

Figure 3.5: City road networks for traffic simulation

Simulated traffic data The simulation of city traffic consists of two parts: road network generation and traffic generation. City road networks are generated from the XML map data downloaded from http://www.openstreetmap.org. Our traffic generator is based on the digital representation of real road networks and the network-based moving object generator of Brinkhoff [4]. A road is a polyline consisting of a sequence of connected line segments. The initial location of a moving object is randomly selected on the road segments. The object then moves along this segment in either direction until reaching crossroads, where it has a 25% chance to stop for several seconds due to the traffic and then continues moving along another randomly selected connected segment.

We assume speed values of the moving vehicles in each road segment follow a random variable X and $X \sim \mathcal{N}(\mu, \sigma^2)$, where $\mathcal{N}$ is the normal distribution, μ and σ are set according to categories of the road segments. We divide the road segments into three categories: C1) freeways/motorways with fastest traffic, C2) primary roads with secondary fastest traffic, and C3) street ways or resi-



(a) Varying N (b) Varying q

Figure 3.6: Overhead for partition update

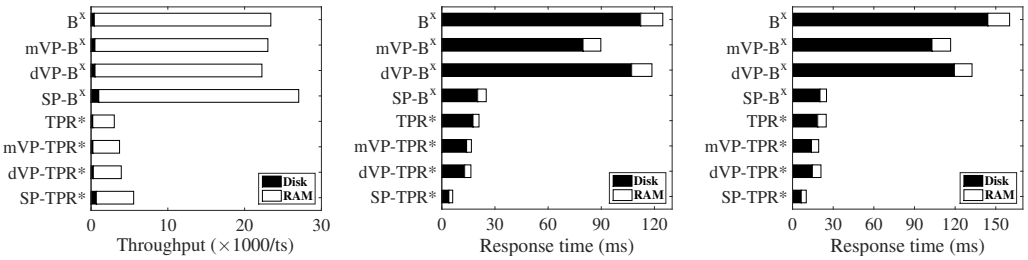
dential roads with slowest traffic. We randomly select the normal distribution parameter μ from a range in terms of m/s for each category: C1) [25, 40], C2) [5, 25], C3) [0, 15]. We set σ=10 m/s for all road segments.

GPS tracking data The SZ dataset contains 100K trajectories of taxis within the urban area of Shenzhen, China. Each trajectory contains a sequence of GPS tracking data with timestamps in a single day. The trajectories are not sampled with equal time intervals and the smallest sampling interval is 15 seconds. The dataset can be accessed at http://mathcs.
emory.edu/aims/spindex/taxi.dat.zip.

Note that we generate 2 minutes traffic data for the simulated datasets, where the distributions of locations and speeds do not change. On the other hand, the SZ dataset has a much longer time span, thus is used to evaluated the temporal factor that leads to distribution changes on locations and speeds.

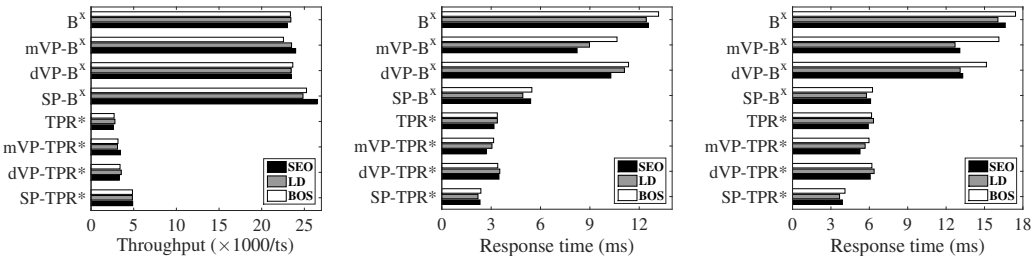
3.3.2 Experimental results

Firstly, we show the execution time of Algorithm 3.2, which is the main overhead for partition updates, with different number of objects (N) and number of speed values (q). Figure 3.6 shows the results, which are consistent with the



(a) Throughput (b) Range query (c) kNN query

Figure 3.7: Disk indexes v.s. RAM indexes



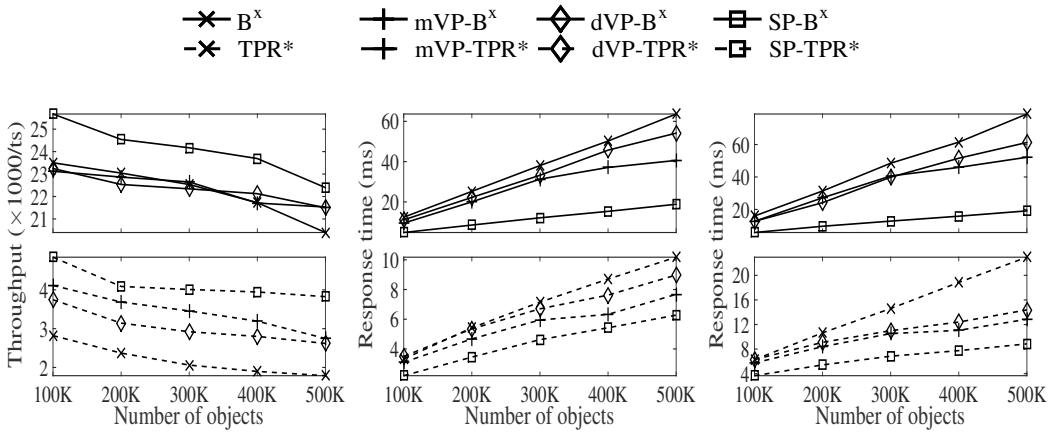
(a) Throughput (b) Range query (c) kNN query

Figure 3.8: Varying datasets

complexity analysis for Algorithm 3.2 in Section 3.2. We find that the execution time is less then 2 second in all settings. Thus the overhead for partition update is reasonably small. We set q equal 50 for the remaining experiments.

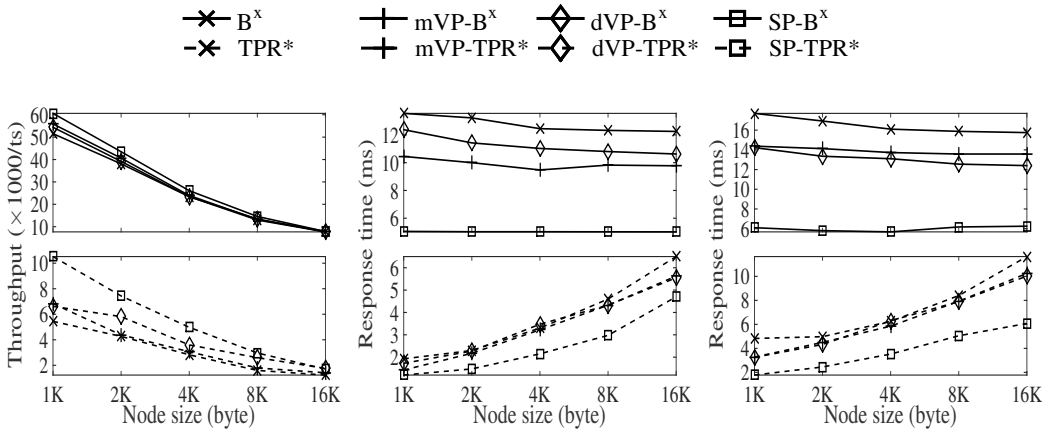
Next we compare the performances between disk indexes and main memory indexes. Figure 3.7(a) through 3.7(c) show results on throughput, range query response time and kNN query response time, respectively. We can see that SP outperforms other methods for both disk and main memory indexes with both B^x-trees and TPR*-trees. Moreover, we found that main memory indexes enjoy much better performance than disk indexes on both throughput and query response time. In the remaining experiments, we report only the results of main memory indexes since we have limited space.

Next we compare the experimental results across three simulated traffic



(a) Throughput (b) Range query (c) kNN query

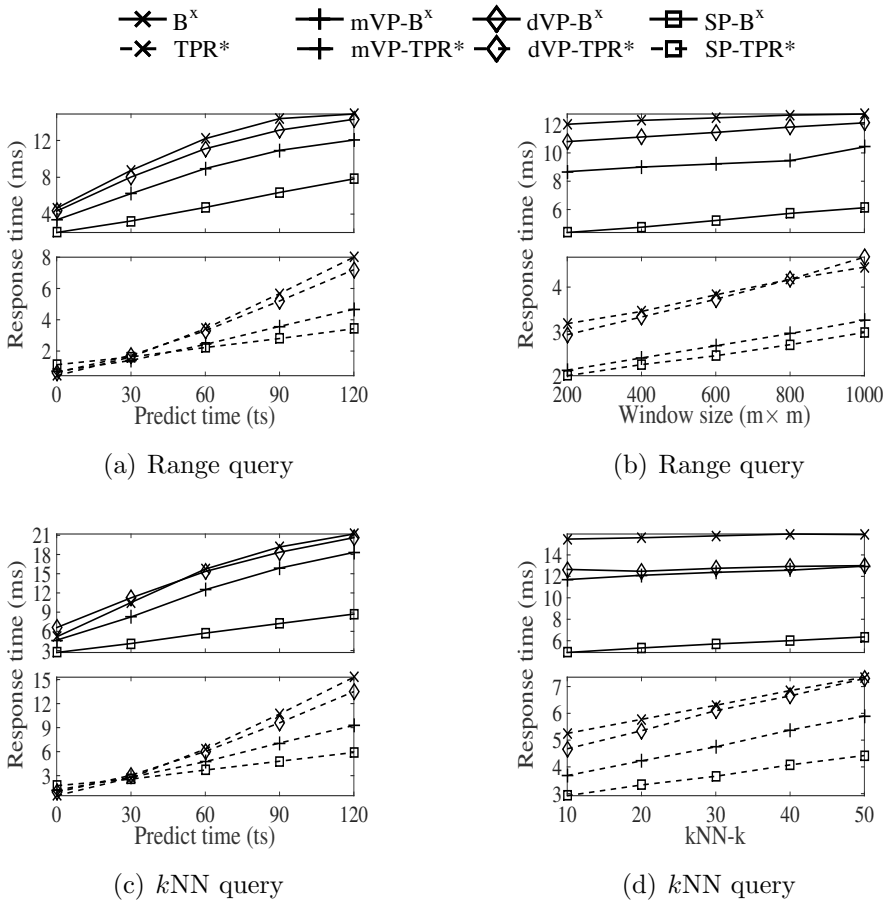
Figure 3.9: Varying number of objects



(a) Throughput (b) Range query (c) kNN query

Figure 3.10: Varying node size

datasets (SEO, LD, and BOS), which are summarized in Figure 3.8. We can see that SP enjoys better performance than other velocity-based partitioning methods as well as the non-partitioning counterparts on a variety of datasets (road networks from Asian, European, and American cities). This is because, as shown in Figure 3.5, road networks for large space domain (10,000×10,000 m^2) usually implies no explicit velocity seeds or DVAs which are used in VMBR-based partitioning and DVA-based partitioning techniques, respectively. More-



(a) Range query (b) Range query

(c) kNN query (d) kNN query

Figure 3.11: Varying query parameters

over, Boston road network has more high speed roads than other city road networks thus nodes in the corresponding indexes expand faster, which makes the BOS dataset has higher query costs than other datasets.

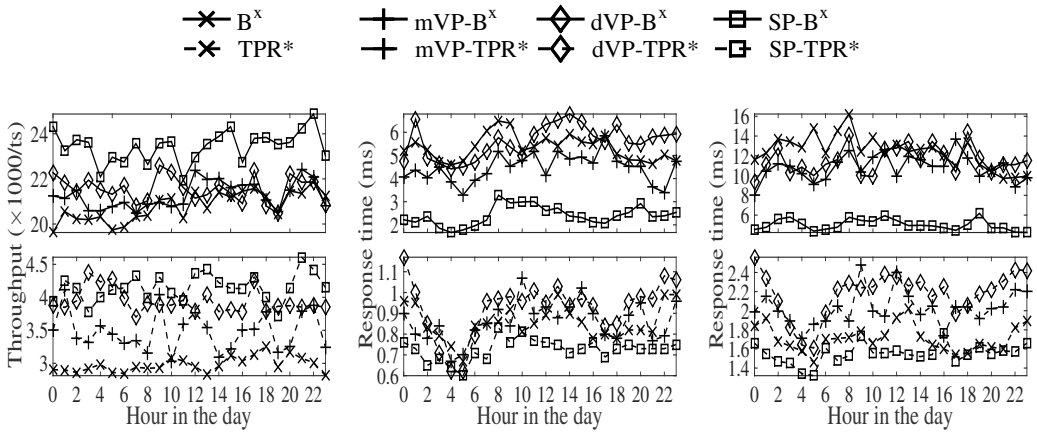
In the next experiment, we vary the number of moving objects from 100K to 500K. Figure 3.9 shows the results about throughput, range query and kNN query. We can see that when the number of objects increases, throughput decreases, query response time increases for both range queries and kNN queries. Moreover, B^x-trees enjoy higher throughput due to the simple update process of B$^+$-tree but lower query utility due to the "false hits" caused by the space-filling curves [18,57]. On the contrary, TPR*-trees have more complicated update op-

erations which makes query more efficient at a sacrifice of throughput. Finally, SP indexes consistently outperform other indexes in all settings.

Next we vary the node size from 1KB to 16KB. Figure 3.10(a) through 3.10(c) show the experimental results. Generally speaking, performance decreases when node size increases, since index nodes with larger sizes require more maintaining and retrieving efforts. However, query performance of B^x-trees is not significantly affected by node size. This is because nodes of B^x-trees store the values computed from space-filling curves, which makes the spatial areas of B^x-tree nodes insensitive to their storage sizes. Note that the experimental results are different from both those for disk indexes, where disk I/O latency dominates the performance [6], and those for main memory indexes with secondary index on object IDs, which enables constant time locating the objects for updates [44]. Finally, SP significantly outperforms other methods in this experiment.

Next we study the impact of query parameters including query predict time, range query window size and kNN-k. The experimental results are summarized in Figure 3.11. Figure 3.11(a) and 3.11(b) show the results about range queries while Figure 3.11(c) and 3.11(d) show those about kNN queries. We can conclude from the figures that, generally speaking, TPR*-trees perform better than B^x-trees and SP outperforms other methods. Moreover, SP gains more advantages when query predict time, query window size, and kNN-k increase.

Finally, we present the results on the real world dataset SZ, which contains information of the taxis in a day long period. Since the distributions of locations and speeds might change during the experiment time, we perform partition updates every 1 hour. The experimental results are summarized in Figure 3.12. We can see that query costs are lowest at early morning, since most cities have



(a) Throughput (b) Range query (c) kNN query

Figure 3.12: Varying hour of the day

least volume of traffic during that time period. We also find that query costs raise at noon and night. This is because the taxis drive faster resulting in higher expanding speeds of the index nodes. The variation of throughput during the day is relatively small. Again, SP significantly and consistently outperforms other partitioning methods and their unpartitioned counterparts.

Chapter 4

Uniform Grid Index in Dual Space

In this chapter, I present the D-Grid, an in-memory dual space grid index for moving objects. Specifically, it indexes moving objects using grid structures in both location and velocity spaces, which improves query performance by almost an order of magnitude. We also propose a lazy deletion and garbage cleaning mechanism that can be applied to both our dual space and existing location space uniform grid based indexes and further improve update performance. Extensive experiments demonstrate that our approach significantly outperforms existing uniform grid based in-memory indexes.

4.1 Indexing with Dual Space Grids

In this section, we introduce the D-Grid, which is an in-memory indexing structure in the dual space for moving object databases.

4.1.1 Structure overview

Intuitions of design

Before we introduce the structure of D-Grid, we first discuss intuitions behind the design. The main purpose of D-Grid is to improve the performance of uniform grid indexes for moving objects on predictive spatio-temporal queries [31, 41, 52]. First, we briefly present how a predictive range query is processed on the uniform grid. A predictive range query Q searches the objects located within a spatial region S, which we call the *query window*, at a future timestamp t_q, which we call the *query prediction time*. Consider the example as in Figure 4.1(a), where the solid rectangle S denotes the query window. p_1 and p_2 (the dots) denote the *indexed locations* of two objects at the *index timestamp*, denoted as t_{idx}, while p'_1 and p'_2 (the circles) denote their locations at the query prediction time t_q. Note that the index stores object locations at the synchronized index timestamp, which are derived from the last updated locations and velocities of the objects. Obviously, both objects should be in the result of this query. To obtain such result, the query window S should be enlarged to S' (the dashed rectangle) to include objects that may move into the query window S at the future time t_q. This is achieved by attaching *enlargement speeds*, v_u, v_d, v_r, and v_l on each side of S. We call the tuple $\{v_u, v_d, v_r, v_l\}$ the *query window enlargement rectangle* ($QwER$), which represents a rectangle in the velocity space. We use the same approach as in [18] to compute $QwER$s (will be explained in Section 4.1.3). The enlarged query window S', computed by expanding S in four directions according to the $QwER$, is the minimum spatial region that can bound the indexed locations of all objects that can possibly be

in the result of query Q. Specifically

$$S'_u = S_u + (t_q - t_{idx}) \times v_u \tag{4.1}$$

where S_u and S'_u denote the upside bound of S and S'. S'_d, S'_r, and S'_l can be calculated similarly.

Uniform grid. Consider a baseline uniform grid approach where the objects are partitioned with grid cells based on their locations. As shown in Figure 4.1(a), let $S^\top$ denote the minimum rectangle with grid cell boundaries that contains S'. We call the set of objects within $S^\top$ the *candidate set*, denoted as $\mathcal{C}$. The actual answer set of query Q, denoted as $\mathcal{A}$, is a subset of $\mathcal{C}$. Given a predictive range query Q, we first compute $S^\top$ and $\mathcal{C}$. We then evaluate whether each object in $\mathcal{C}$, which we call a *candidate object*, will move into S at timestamp t_q based on its indexed location and velocity. Since no disk I/O is invoked in main memory environment, the processing time of query Q is determined by the total number of candidate objects. We assume that the moving objects are uniformly distributed in the predefined location space (otherwise we can always partition the location space into uniform subregions [52]) and let σ denote the density of objects. The number of candidate objects is $|\mathcal{C}| = \sigma|S^\top|$, where $|S^\top|$ denotes the spatial size of $S^\top$. Let τ_0 denote the processing time for evaluating whether an object moves into S at timestamp t_q. The processing time for answering query Q using the uniform grid can be estimated by

$$\tau = \tau_0 \sigma |S^\top| \tag{4.2}$$

Dual space grid. A potential drawback of the above approach is that $QwER$ considers the "maximum" velocity of the objects when computing the

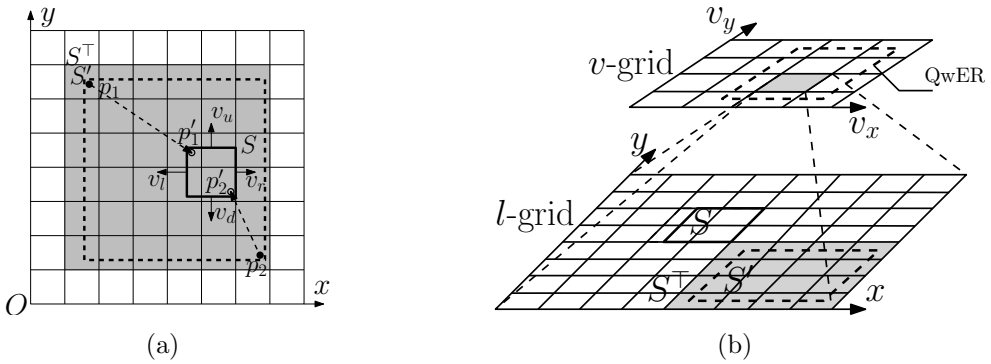


Figure 4.1: Query window enlargement

enlarged query window. Hence some candidate objects in the enlarged query window with small velocity values might not move into the query window at t_q. Motivated by this, we can partition the objects based on their velocities and model the predefined velocity space as another uniform grid, which we call the v-grid, and associate each v-grid cell with a location grid (l-grid). In such structure, as shown in Figure 4.1(b), objects within the same v-grid cell move with similar velocities restricted by the v-grid cell boundaries. Given a predictive range query Q, we first compute the $QwER$. Then for each v-grid cell that intersects with the $QwER$, we perform range search on the enlarged query window in the corresponding l-grid. Let S'_i denote the enlarged query window on the l-grid associated with the i^{th} v-grid cell that intersects with $QwER$ and $S_i^\top$ the corresponding area with l-grid cell boundaries. Let σ_i denote the density of objects in $S_i^\top$, thus the processing time for answering Q can be estimated by

$$\tilde{\tau} = \tau_0 \sum_{\forall i} \sigma_i |S_i^\top| \tag{4.3}$$

According to Equations (4.2) and (4.3) and the facts that $\sum_{\forall i} \sigma_i = \sigma$, and $|S_i^\top| \leqslant |S^\top|$, we have $\tilde{\tau} \leqslant \tau$. Thus, query processing time is reduced by partitioning the velocity space into the v-grid. Intuitively, by grouping objects with similar

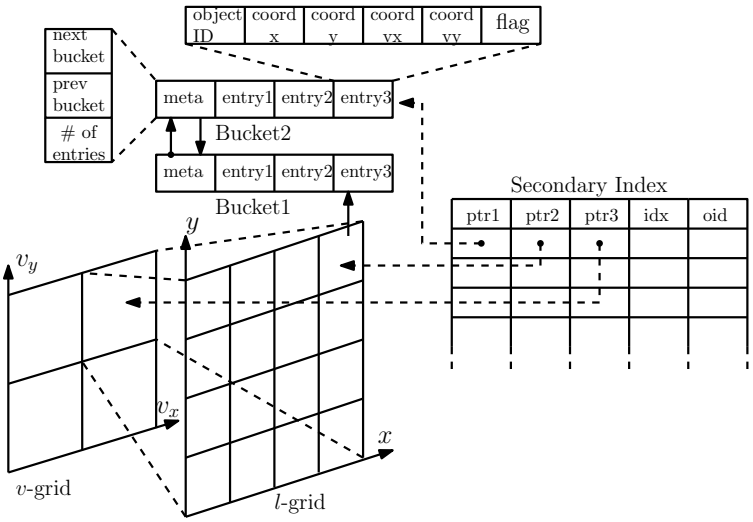


Figure 4.2: Structure of D-Grid

velocities together, the corresponding enlarged query windows are significantly smaller, which reduces the overall candidate set for query processing. Based on this observation, we propose the D-Grid that combines the v-grid and the l-grid.

The structure of D-Grid

Figure 4.2 depicts the structure of D-Grid, which consists of three parts: the v-grid, the l-grid, and the secondary index. As mentioned in Section 4.1.1, the v-grid is a uniform grid on the velocity space and each cell in the v-grid corresponds to an l-grid, where each l-grid cell points to a double-linked list of buckets that stores the object data. Each bucket consists of data entries and a meta data field that contains the current number of entries in the bucket and two pointers that link to the next and previous buckets, respectively, when they exist. The purpose of using double-linked lists instead of singly linked lists is to support the garbage cleaning operation (see Section 4.1.2). Data entries in the buckets contain not only location but also velocity information about the

moving objects. An additional attribute $flag$ stored in the data entries is used for the lazy deletion and garbage cleaning mechanism, which will be discussed in Section 4.1.2 in detail. Similar to u-Grid [44], we employ a secondary index on *oid* (object ID) to provide constant time accessing on the object data for update operations. As illustrated in Figure 4.2, the secondary index for D-Grid contains three pointers with $ptr1$, $ptr2$, and $ptr3$ pointing to the bucket, the l-grid cell, and the v-grid cell, respectively. The attribute idx stores the in-bucket position of the corresponding data entry.

Time partition

Given a predictive range query, according to Equation (4.1), the query window enlargement is determined by not only the $QwER$ but also the index timestamp. Apparently, the more up-to-date the index is, the less the query window enlarges. Thus, we apply the *time partition* technique, which was proposed in [18], on D-Grid. Specifically, the time axis is partitioned into intervals of Δt, which denotes the maximum allowed duration in-between two updates of any object, and each such interval is sub-partitioned into equal-length *phases*. Each phase is associated with a *label timestamp*, $t_{lab} = \frac{k}{n}\Delta t$, where k is the phase ID. Note that the index timestamp of each phase refers to the corresponding label timestamp, i.e. $t_{idx} \equiv t_{lab}$. Each object update is associated with a nearest future label timestamp and its location at the label timestamp is calculated and stored in the corresponding phase. Note that only the most recent $n + 1$ phases are kept in main memory and that the outdated objects are forced to update after Δt timestamps since their last updates. Figure 4.3 shows an example of time partition when $n = 2$. Specifically, updates issued in the time ranges $(0, \frac{1}{2}\Delta t]$ and $(\frac{1}{2}\Delta t, \Delta t]$ are inserted into the first two phases

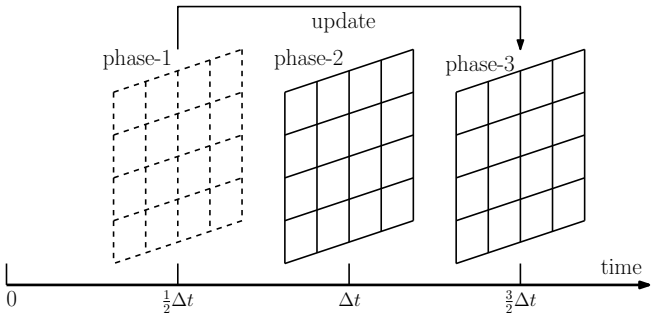


Figure 4.3: Time partition

with label timestamps $\frac{1}{2}\Delta t$ and Δt, respectively. After timestamp Δt, phase-1 expires and the objects that update between timestamp Δt and $\frac{3}{2}\Delta t$ will be deleted from phase-1 and inserted into phase-3. Thus phase-1 will be empty and be abandoned at timestamp $\frac{3}{2}\Delta t$. With time partition, indexed locations of the objects can only be outdated by a maximum of Δt. The method of choosing n is described in [18] and we set $n = 2$ in this paper.

4.1.2 Updates

In this section, we introduce the update algorithms of D-Grid as well as the lazy deletion and garbage cleaning mechanism.

Local and non-local updates

Two different scenarios are considered for the update operation: *local update* and *non-local update*. When an object issues an update, two fields $ptr2$ and $ptr3$ in the secondary index are used to determine whether the object belongs to the same grid cell (v-grid and l-grid) as its previous update. If so, local update is performed by simply copying the updated data to the corresponding data entry located by $ptr1$ and idx. Otherwise, non-local update is performed instead, which first deletes the object from its previous position and then inserts

Algorithm 4.1: Update

```
   input  : o (the object to be updated whose ID is oid)
   /* check the grid cells for the coming update                    */
1  VC_old ← SI_oid.ptr3, LC_old ← SI_oid.ptr2;
2  BUK ← SI_oid.ptr1, IDX ← SI_oid.idx;
3  VC_new ← new v-grid cell of o;
4  LC_new ← new l-grid cell of o;
5  if VC_old = VC_new and LC_old = LC_new then
       /* perform local update                                      */
6      BUK[IDX].coords ← o.coords;
7  else
       /* perform non-local update                                  */
8      if percentage of invalid entries in (VC_old, LC_old) < λ then
           /* mark the data entry as invalid                        */
9          BUK[IDX].flag ← false;
10     else
           /* Garbage cleaning                                      */
11         Garbage Cleaning(VC_old, LC_old);
12     insert o into cell (VC_new, LC_new);
13     update SI_oid;
```

the updated data into the new grid cell. The secondary index also needs to be updated with non-local updates. Similar to u-Grid [44], update operations ensure that all except the first bucket of the l-grid cells are full. Specifically, a new object is always inserted at the end of the first bucket. If the first bucket is full, a new bucket is allocated and becomes the first bucket. Moreover, a deletion operation always moves the last object of the first bucket to the position of the deleted object. When the first bucket becomes empty, it will be removed and the second bucket becomes the first if it exists otherwise the bucket list will be replaced by a *null* pointer.

Lazy deletion and garbage cleaning

The local update and insertion operations are optimized since they involve only unavoidable data copies, which is the theoretically minimum cost to update the object data in any indexing structure. Next, we introduce the lazy deletion and garbage cleaning mechanism that further optimizes the deletion operations involved in non-local updates.

Theoretically, an optimal approach for the deletion operation is to mark the objects to be deleted as invalid instead of actually erasing them from the index. However, this will not only increase the storage size but also hurt query performance, since invalid objects need to be filtered out while processing queries. In order to improve update cost without sacrificing query performance, we propose the LDGC mechanism, which reduces the amortized cost of deletion operations in generic uniform grid based indexes. In our approach, the object to be deleted is marked as invalid by turning off the $flag$ attribute of the corresponding data entry if the percentage of invalid objects in the corresponding l-grid cell is below a predefined threshold, otherwise the garbage cleaning operation is performed on this cell. Figure 4.4 illustrates the mechanism of garbage cleaning, where the white (gray) rectangles represent the valid (invalid) data entries and the bucket surrounded by a dashed rectangle is deleted during garbage cleaning. Basically, valid data entries are copied to positions with invalid entries from the end of the bucket list, which are illustrated by dashed arrows in Figure 4.4.

Next we briefly analyze the benefit of the LDGC mechanism. Let n denote the total number of objects in a certain l-grid cell and λ the threshold that triggers garbage cleaning, thus the number of valid and invalid data entries are $(1-\lambda)n$ and λn, respectively, when garbage cleaning is triggered. Without loss of generality, we assume the positions of valid entries are uniformly distributed

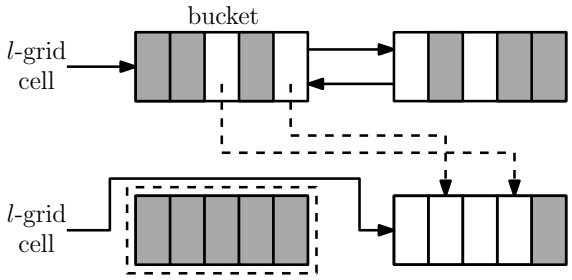


Figure 4.4: Garbage cleaning

Algorithm 4.2: Garbage Cleaning

```
   input : (VC, LC) (grid cell to be cleaned)
1  (BUK1, IDX1) ← first entry in last bucket;
2  (BUK2, IDX2) ← last entry in first bucket;
3  while (BUK1, IDX1) ≠ (BUK2, IDX2) do
       /* find the next invalid entry                    */
4      (BUK1, IDX1) ← next invalid entry;
       /* find the next valid entry                      */
5      (BUK2, IDX2) ← next valid entry;
6      BUK1[IDX1] ← BUK2[IDX2];
7      BUK2[IDX2].flag ← false;
8      update SI_{BUK1[IDX1].oid};
9  delete empty buckets;
```

in the buckets and thus the expected number of valid entries in the first λn positions is $(1-\lambda)\lambda n$, which equals the expected number of data copies when LDGC is in effect. On the other hand, if LDGC is not applied, the number of data copies is λn (one for each deletion). For example in Figure 4.4, 6 data copies are performed with regular deletion strategy, since there are 6 invalid entries, while the number of data copies decreases to 2 with LDGC. Let μ_0 denote the time cost for one data copy, thus LDGC can save $\mu_0(\lambda n - (1-\lambda)\lambda n) = \mu_0\lambda^2 n$ time. On the other hand, traversing the data entries takes $\nu_0 n$ time, where ν_0 denotes the time cost for fetching one entry from the buckets. Let $\Gamma = \mu_0\lambda^2 n - \nu_0 n$, thus, when $\Gamma > 0$, i.e. $\lambda > \sqrt{\frac{\nu_0}{\mu_0}}$, LDGC gains benefits. We will also empirically show the benefits of LDGC in Section 4.2.

Algorithm 4.1 summarizes the main steps for the update operation in D-Grid. SI represents the secondary index. VC and LC represent the v-grid cell and the l-grid cell, respectively. BUK and IDX represent the bucket and the in-bucket position, respectively. After receiving an update, Algorithm 4.1 first computes the old and new grid cells where the object is located (lines 1-4). If the old and new grid cells are the same, local update is performed (line 6), otherwise non-local update is performed (lines 8-13). In non-local update, if the number of invalid entries in the corresponding grid cell is below the threshold λ, we simply mark the entry as invalid (line 9), otherwise we perform the garbage cleaning operation (line 11). The method *Garbage Cleaning* is described in Algorithm 4.2. Specifically, we use two pointers (BUK_1, IDX_1) and (BUK_2, IDX_2) to locate the invalid and valid data entries involved in each data copy, respectively. (BUK_1, IDX_1) and (BUK_2, IDX_2) iterate backwardly and forwardly from the last and first bucket, respectively, and point to the corresponding entries for the data copy (lines 4-5). The iterating is accomplished with the help of the double-linked bucket list, mentioned in Section 4.1.1. After the data copy is finished (lines 6-7), we turn off the $flag$ of the data entry pointed to by (BUK_2, IDX_2) (line 7) and update the corresponding entry in the secondary index (line 8). Finally, the empty buckets, that contain only invalid entries, are deleted (line 9).

4.1.3 Query processing

In this section, we introduce the algorithms for processing predictive range queries and k nearest neighbor (kNN) queries in D-Grid, respectively. Figure 4.5 illustrates the workflow of query processing in D-Grid.

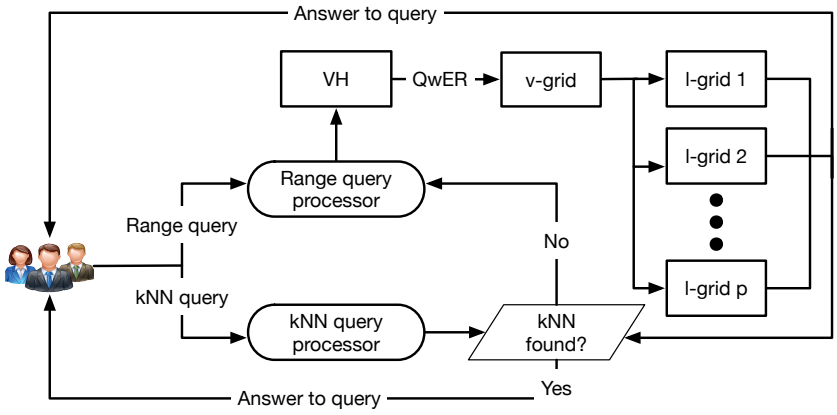


Figure 4.5: Query processing in D-Grid

Range queries

A predictive range query searches objects within a query window S (a square area with side length $qLen$), at a (future) query prediction time t_q. Since objects are continuously moving, the queries must be performed with enlarged query windows at the index timestamps, which are determined by $QwER$s described in Section 4.1.1, in order to include all possible answers of the queries. As introduced in [18], we first set the $QwER$ with the maximum speed of all objects in each direction. Then, the final enlargement speeds are computed with the aid of the *velocity histogram*, which is a 2-dimensional grid that captures the maximum projections of the object velocities onto each direction in each l-grid cell.

After receiving the range query, our system first computes the $QwER$ using the velocity histogram and then generates the sub-queries with enlarged query windows according to each v-grid cell that intersects with the $QwER$. After the sub-queries are processed, the sub-query results are combined and returned to the user. The pseudo code for predictive range query is shown in Algorithm 4.3, which takes the query Q as input and returns the answer set $\mathcal{A}$ as output.

Algorithm 4.3: Range query

```
   input  : Q (range query)
   output: A (answer set)
   /* initialization                                          */
1  A ← ∅, QwER ←VH(Q);
2  foreach VC ∈ QwER do
       /* compute the enlarged window                        */
3      QwER* ← QwER ⋂ VC;
4      S^⊤ ← enlarge(Q, QwER);
       /* compute the answer to Q                            */
5      foreach o located within S^⊤ do
6          if o is answer to Q then
7              A.add(o);
8  return A;
```

Algorithm 4.3 first initializes the answer set (line 1) and computes the $QwER$ of query Q via the velocity histogram VH [18] (line 2). Then it processes the sub-queries according to each v-grid cell, where the enlarged query window $S^{\top}$ is first computed (lines 3-4) and objects located within $S^{\top}$ are then retrieved and filtered to get the final answer set (lines 5-7).

k nearest neighbor queries

A predictive k nearest neighbor (kNN) query with a query point at $loc = (q_x, q_y)$, which is a coordinate in the location space, searches the k objects that no other objects are nearer to the query point at the (future) query prediction time t_q. To answer the kNN query, we iteratively perform range queries around the query point with gradually increasing window sizes until the k nearest neighbors are found. The expected distance from the query point to its k^{th}

Algorithm 4.4: k nearest neighbor query

```
   input  : Q (kNN query)
   output: A (answer set)
   /* initialization                                                    */
 1 A ← ∅, D_k ← compute D_k, MH ← create max heap;
 2 RQ ← new range query (Q.loc, D_k);
 3 while RQ.qLen < D do
       /* perform range query                                          */
 4     A_0 ← Range Query(RQ);
 5     foreach o ∈ A_0 do
           /* replace the top of MH when necessary                     */
 6         if MH.size < k then
 7             MH.insert(o);
 8         else
 9             o' ← MH.top;
10             if dist(Q.loc, o)¡dist(Q.loc, o') then
11                 MH.delete(o'), MH.insert(o);
12     if MH.size = k and dist(Q.loc, MH.top) ⩽ RQ.qLen then
13         break;
14     RQ.qLen ← RQ.qLen + D_k;
15 A.addAll(MH);
16 return A;
```

nearest neighbor can be estimated as

$$D_k = \frac{2}{\sqrt{\pi}} \left[1 - \sqrt{1 - \left(\frac{k}{N} \right)^{\frac{1}{2}}} \right] \tag{4.4}$$

where N represents the total number of objects [50]. The query window of the $i^{th} (i \geqslant 1)$ range query, denoted as S_i, is a square with side length $i \cdot D_k$. During the execution of the kNN query, a max-heap is used to keep track of the k nearest neighbors as well as the distance from the query point to the k^{th} nearest neighbor at the current iteration.

Algorithm 4.4 summarizes the approach for processing predictive kNN queries

in D-Grid. This algorithm first initializes the answer set ($\mathcal{A}$), the estimated distance of the k^{th} nearest neighbor (D_k), the max heap (MH), and the range query (RQ) (lines 1-2) and then enters the loop (lines 3-14) that repeats when the side length of the query window $RQ.qLen$ is smaller than that of the predefined location space (D). During each iteration of the loop, RQ is first executed and the answers are stored in $\mathcal{A}_0$. Then for each object in $\mathcal{A}_0$, if the size of MH is smaller than k, we directly insert it into MH. Otherwise, we compare the distances (to the query point) from o and the top element o' in MH, which is the k^{th} nearest neighbor in the previous iteration. If o is nearer than o', we delete o' from MH and insert o into MH. After all objects in $\mathcal{A}_0$ are processed, we check whether the termination condition is satisfied (line 12). Then, we either exit the loop or increase $RQ.qLen$ and move forward to the next iteration. Finally, the objects in MH are added to $\mathcal{A}$ as the query result (line 15).

4.2 Experimental Study

In this section, we conduct experiments on a variety of datasets and parameters to evaluate the performance of D-Grid and compare with other state-of-the-art uniform grid based indexing structures. Both update time and query processing time are evaluated in the experiments. All algorithms are implemented with C++ and all experiments are performed with 2.6 GHz Intel Core i7 CPU and 16GB RAM in OSX 10.11 operating system. The parameters in D-Grid will be tuned in Section 4.2.2 and Table 4.1 shows the tuning results. The experimental settings are displayed in Table 4.2 where the default settings are boldfaced.

Table 4.1: D-Grid parameters

l-grid cell length (m)	2000
v-grid cell length (m/s)	50
Bucket size (byte)	2048
Garbage cleaning threshold	0.8

Table 4.2: Experimental settings

Space domain (m×m)	**100,000×100,000**
Maximum speed (m/s)	10, 20, 30, ..., **100**
Number of objects	100K, 200K, 300K, ..., **1M**
Window length (m)	10, 100, **1000**, 10,000
kNN - k	1, 10, **100**, 1000
Query prediction time (ts)	0, 30, **60**, 90
Dataset	**Uniform**, Gaussian, RN, GPS

4.2.1 Dataset description

We first describe the datasets used in the experiments.

Uniform: In the synthetic uniform dataset, the objects are uniformly distributed in the predefined location space and moves with random speeds at arbitrary directions.

Gaussian: In the synthetic Gaussian dataset, the objects travel between the predefined hotspots in the location space. The locations of the objects are Gaussian distributed around the hotspots and the velocities are random. We use a Gaussian dataset with 10 hotspots. Figure 4.6 illustrate a snapshot of the objects in this dataset.

Road network: The road network (RN) dataset is generated with the same method described in Chapter 3. Figure 4.7 shows a part of Los Angeles road network, which is the underlying road network of the RN dataset.

GPS: The real GPS dataset contains one million trajectories of taxis within the urban area of Shenzhen, China. Each trajectory contains a sequence of GPS tracking data with timestamped locations in a single day. The trajectories are

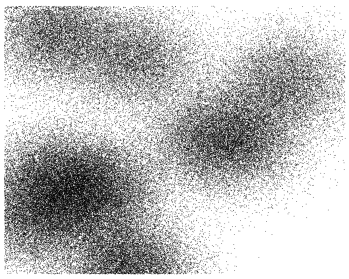


Figure 4.6: Gaussian

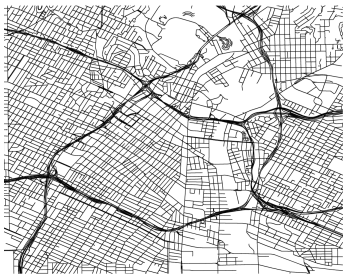
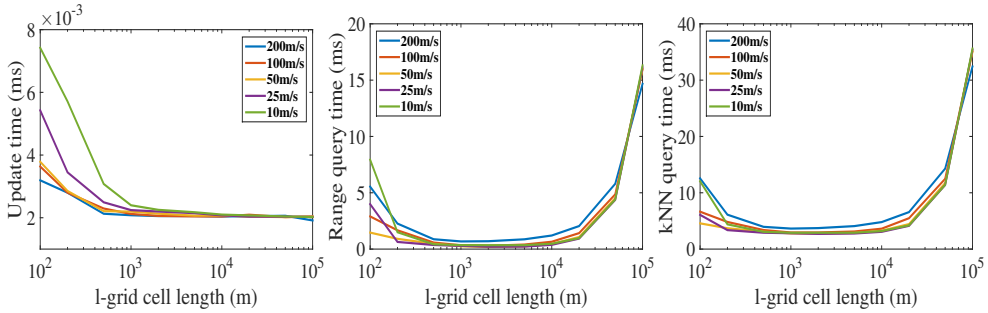


Figure 4.7: Road network



(a) Update (b) Range query (c) kNN query

Figure 4.8: Vary grid cell length

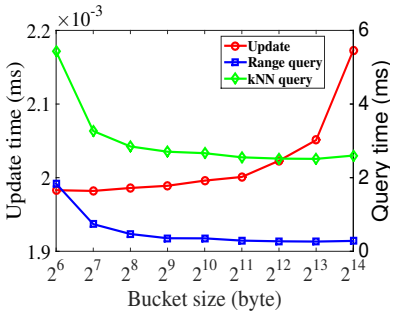


Figure 4.9: Vary bucket size

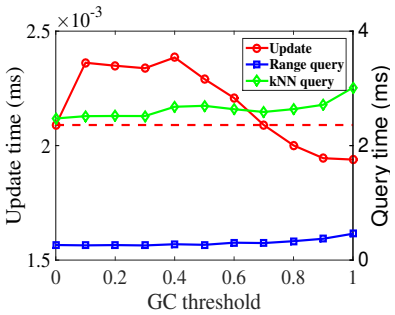
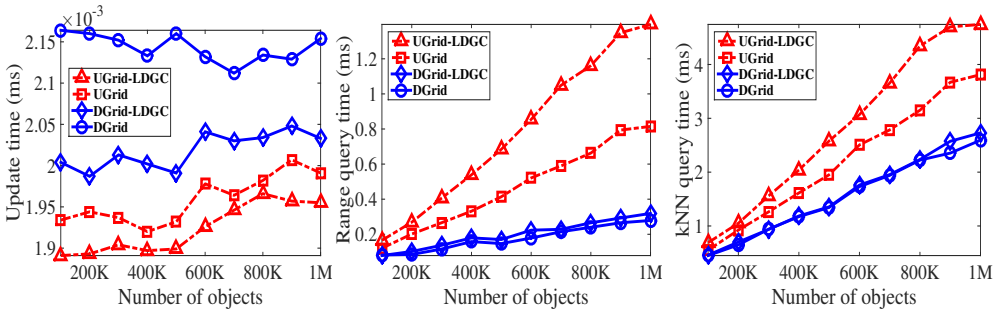


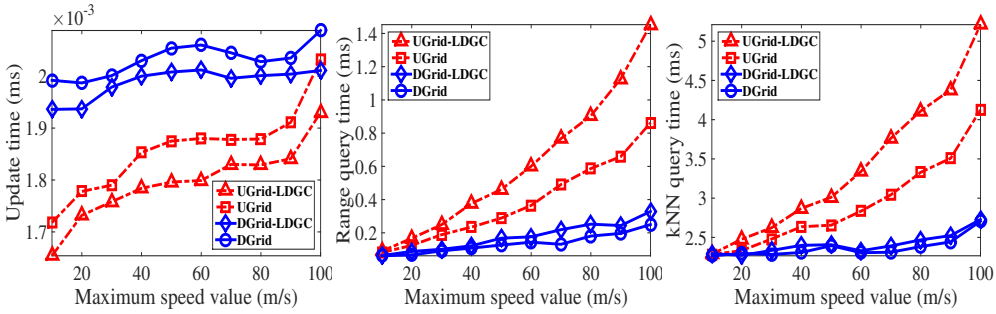
Figure 4.10: Vary GC threshold

not sampled with equal time intervals and the smallest sampling interval is 15 seconds. The dataset can be downloaded at http://mathcs.emory.edu/aims/spindex/gps_1M.dat.zip. Note that GPS records in this dataset are mapped to a 100,000 × 100,000 location space.



(a) Update (b) Range query (c) kNN query

Figure 4.11: Vary number of objects



(a) Update (b) Range query (c) kNN query

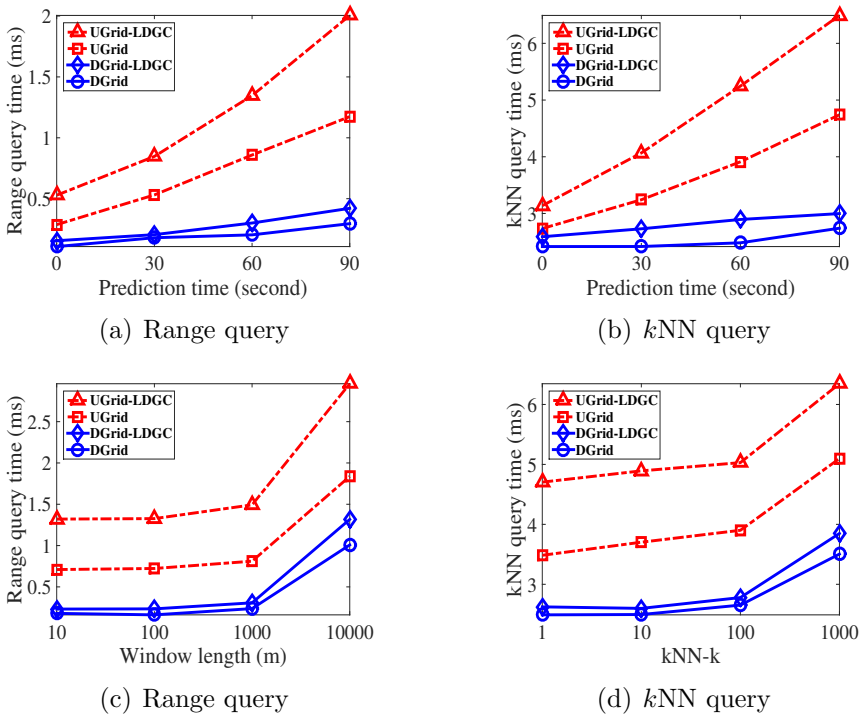
Figure 4.12: Vary maximum speed value

4.2.2 Evaluation of our methods

We first evaluate the performance of D-Grid, with different settings of the parameters including grid cell length (for both v-grid and l-grid), the bucket size and the GC threshold λ.

Impact of grid cell length

In this experiment, we find the optimal settings for grid (both l-grid and v-grid) cell lengths. Note that LDGC is not applied in this and next experiments. Figure 4.8 summarizes the experimental results where the x-axis represents the cell length of l-grid while each line in the figures represents a setting of v-grid cell length. As shown in Figure 4.8(a), update costs reduce when l-grid cell



(a) Range query (b) kNN query

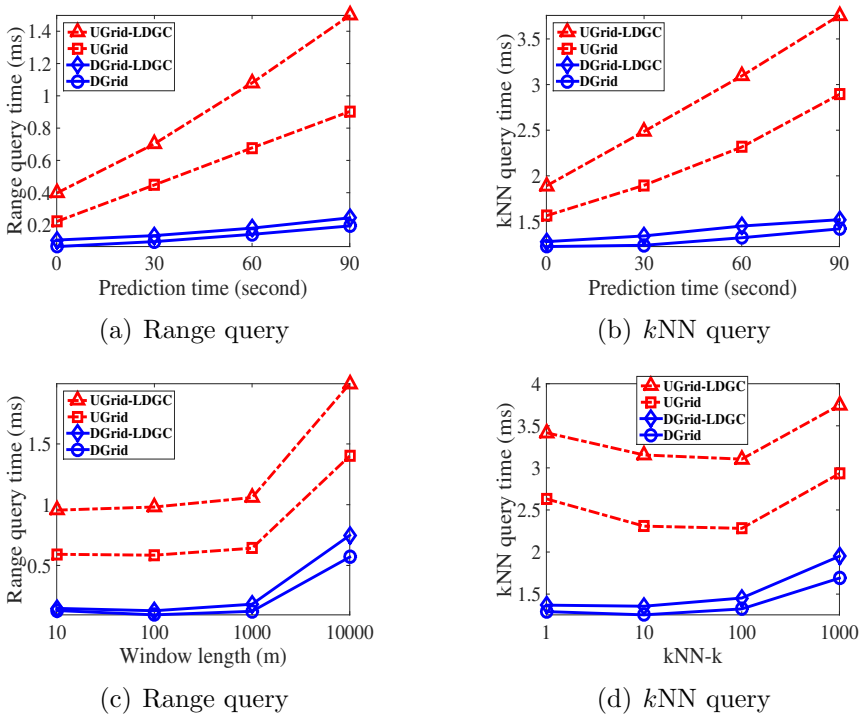
(c) Range query (d) kNN query

Figure 4.13: Vary query parameters on uniform dataset

length increases. This is because local updates happen more frequently with larger l-grid cells. Figure 4.8(b) and 4.8(c) show the results about range queries and kNN queries, respectively. Both range query and kNN query obtain highest performance when the l-grid is between 1000m and 10,000m. When the l-grid cells are too large, the queries tend to be linear search and when they are too small, the overhead for traversing the cells themselves increases dramatically. Similarly, the v-grid cell length of 50m/s leads to the best performance for overall performance. According to the experimental results, we set 2000m and 50m/s as the default l-grid and v-grid cell lengths, respectively.

Impact of bucket size

In this experiment, we study the impact of bucket size. Figure 4.9 shows that query performance benefits from large bucket size. This is because large buckets



(a) Range query (b) kNN query

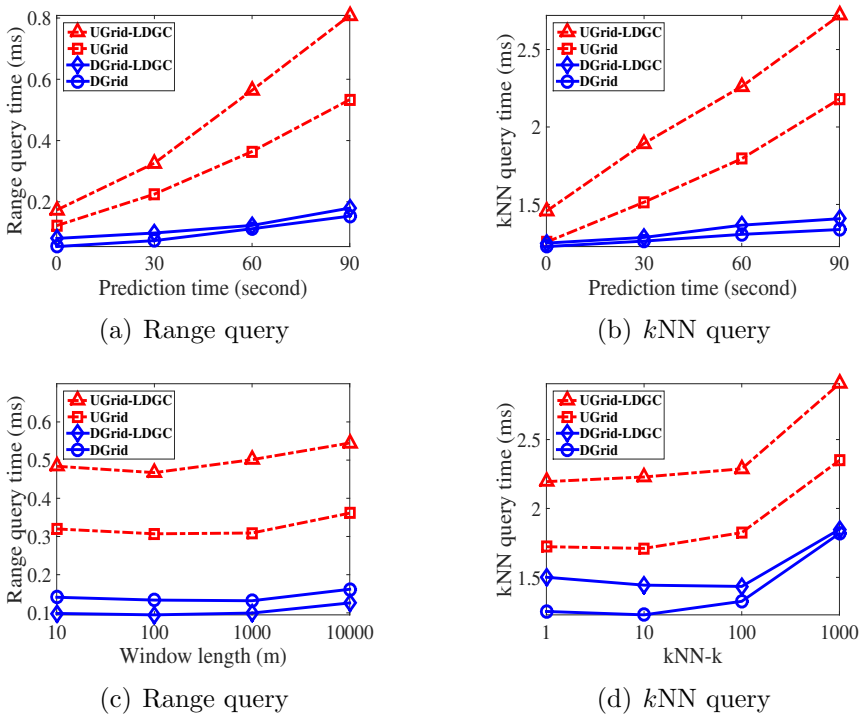
(c) Range query (d) kNN query

Figure 4.14: Vary query parameters on Gaussian dataset

increase data access locality and enable more effective data fetching by CPU caches [9]. However, as shown in Figure 4.9, large buckets lead to worse update performance. Considering the trade-off between update and query costs, we select 2048 bytes as the default bucket size.

Impact of GC threshold

In this experiment, we demonstrate the benefit of LDGC by varying the garbage cleaning (GC) threshold, λ. Note that LDGC is not applied when $\lambda = 0$. The results are shown in Figure 4.10. We can see that query performances reduce as λ increases since invalid data entries need to be filtered out while answering queries. We also find that when LDGC is applied, larger values of λ lead to lower update costs and the costs are below that without LDGC when $\lambda > 0.7$. This observation is consistent with the analysis in Section 4.1.2. Considering



(a) Range query (b) kNN query

(c) Range query (d) kNN query

Figure 4.15: Vary query parameters on road network dataset

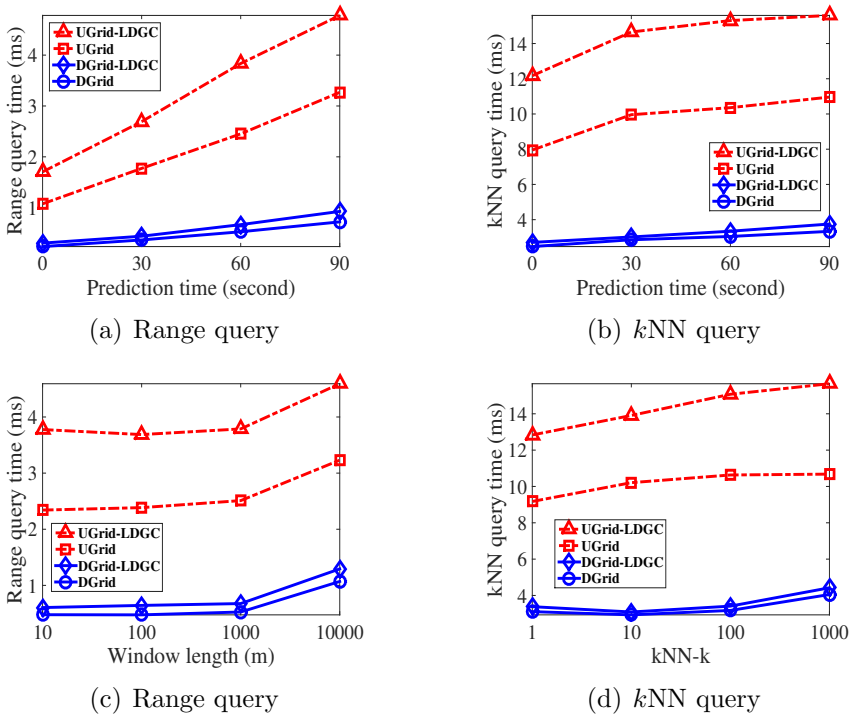
the impact of GC threshold on update and query performances, we select 0.8 as its default value.

4.2.3 Comparison with other methods

Finally, we compare our D-Grid with u-Grid, which is the state-of-the-art uniform grid indexing structure in single thread environment. The parameters in u-Grids are consistent with those in the original paper [44]. We also compare the performances of these methods with or without LDGC.

Impact of number of objects

Figure 4.11 shows the experimental results with varying number of objects. We can see that costs for both range queries and kNN queries grow when the



(a) Range query (b) kNN query

(c) Range query (d) kNN query

Figure 4.16: Vary query parameters on GPS dataset

total number of objects increases while update performance shows little changes with different settings. Moreover, we find that the LDGC mechanism improves update performances for both u-Grid and D-Grid with a trade-off on query performances. Finally, D-Grids significantly outperform u-Grids in terms of query processing but drop on update performance.

Impact of maximum speed value

In this experiment, we vary the maximum speed value of the objects. Figure 4.12 summarizes the results. From Figure 4.12(a) we conclude that the higher speeds of the objects, the higher update costs for both u-Grids and D-Grids. This is because high speed objects are more likely to incur non-local updates, which are more expensive. Figure 4.12(b) and 4.12(c) show that high speed values result in high query costs, since the query windows enlarge when speeds

of the objects increase. Again, LDGC makes the indexes more update efficient but slower for query processing and D-Grids enjoy lower query costs but higher update costs than u-Grids.

Impact of query parameters

Finally, we compare the performances under different query parameters, including query prediction time, range query window length and kNN-k on all the four datasets. Update performances are not evaluated in this experiment, since query parameters do not affect the update processing. Figure 4.13 through 4.16 show the experimental results for each dataset, respectively. We can conclude from the results that D-Grid significantly and consistently outperforms u-Grid in all settings of query parameters on a variety of datasets.

Chapter 5

Markov Chain Based Pruning

In this chapter, I present a pruning mechanism that reduces the candidate set for predictive range queries based on high order Markov chain models learned from historical trajectories. The key to our approach is to devise compressed representations for sparse multi-dimensional matrices, and leverage efficient algorithms for matrix computations. Experimental evaluations show that our approach significantly outperforms other pruning methods in terms of efficiency and precision.

5.1 Preliminaries and Problem Definition

We introduce some preliminary definitions and our problem setting in this section. Table 5.1 lists the notations used throughout this chapter.

5.1.1 Trajectory and path

The predefined space domain is partitioned into uniform grid cells and each cell has an identifier, which is the sequence number of the grid cell in the

Table 5.1: Notations

O	moving object	$\mathcal{O}$	set of all objects
N	number of states	Q	predictive range query
$\mathbf{q}$	query vector	R	query window
t_c	current time	t_q	prediction time
k	order of Markov chain	M_k	Markov transition matrix
s_i/i	state	$\mathcal{S}$	state space
ρ_k^t	state matrix	$\mathcal{I}_h^t$	h-backward path
ϕ	diagonal	ξ	offset of diagonal
$\mathcal{D}$	diagonal matrix	Ξ	offset array
$\mathbf{d}$	diagonal array	ι	shift of projection
$\mathcal{T}$	trajectory	$\mathcal{I}$	path/coordinate
ω_ϕ	valid range	H	number of phases

underlying *space-filling curve* (see Section 5.2.2). We call the grid cells *states* and the set of all states, $\mathcal{S} = \{s_0, s_1, \ldots, s_{N-1}\}$, the *state space*, where $N = |\mathcal{S}|$ denoting the total number of states. In this chapter, we alternatively denote s_i with its identifier i, $\forall 0 \leqslant i < N$, when there is no ambiguity. A *trajectory* of an object $O \in \mathcal{O}$ is defined as a sequence of timestamped locations $\mathcal{T} =< O_0, O_1, \ldots, O_t, \cdots >$, where $O_t = ((x, y), t)$ denotes that O is located at (x, y) at time t. As each location is associated with a state, we call a sequence of states of O, denoted as $\mathcal{I} =< i_0, i_1, \ldots, i_t >$, a *path* of O. Figure 5.4 shows examples of paths on the left. We refer to the path of O in the h timestamps from $t - h + 1$ to t as an *h-backward path* ending at t and denote it as $\mathcal{I}_h^t$, i.e. $\mathcal{I}_h^t =< i_{t-h+1}, i_{t-h+2}, \ldots, i_t >$. Particularly, we call an h-backward path ending at the current time t_c, $\mathcal{I}_h^{t_c}$, the *base h-backward path*.

5.1.2 Predictive range query

We define a predictive range query as follows.

Definition 5.1 (Predictive range query)**.** Given a set of moving objects, $\mathcal{O}$,

with their recent trajectories, $\mathcal{T}$, a spatial region R, and a prediction time t_q, the predictive range query, $Q(R, t_q)$, returns the set of objects in region R at time t_q.

Generally speaking, a predictive range query is processed in two steps: pruning and verification [18, 41, 48, 49]. The pruning step aims to filter out non-qualifying objects that will probably not be in the query result, via a fast pruning mechanism. Candidate objects surviving the pruning step are fed to the verification step that computes their probabilities of satisfying the query predicate. We call the base h-backward paths of the candidate objects the *candidate paths*. The verification step dominates the overall execution time for query processing according to our experimental results (see Section 5.4). Hence it is important to have effective pruning mechanisms to reduce the candidate set.

5.1.3 The Markov chain model

In most real world scenarios, like vehicles in a city road network, objects' (future) paths after t_c are usually uncertain but follow certain patterns. We use time-homogeneous Markov chains [27] to capture such mobility patterns.

Definition 5.2 (Order-k Markov chain)**.** A stochastic process o_t, is called an order-k Markov chain if and only if $\forall i_t, i_{t-1}, \ldots, i_0 \in \mathcal{S}$

$$\begin{aligned} &P(o_t = i_t | o_0 = i_0, \ldots, o_{t-k} = i_{t-k}, \ldots, o_{t-1} = i_{t-1}) \\ =&P(o_t = i_t | o_{t-k} = i_{t-k}, \ldots, o_{t-1} = i_{t-1}) \end{aligned} \tag{5.1}$$

We call Equation (5.1) the *local Markov property* of order-k Markov chains. The conditional probability $P(i_t | i_{t-k}, \ldots, i_{t-1}) = P(o_t = i_t | o_{t-1} = i_{t-1}, \ldots, o_{t-k} =$

i_{t-k}) is the *transition probability*, which indicates the probability for an object moving to state i_t given its previous k states $i_{t-k}, \ldots, i_{t-1}$. The $\underbrace{N \times N \times, \ldots, \times N}_{\text{k+1 N's}}$ multi-dimensional matrix M_k is the transition matrix of the order-k Markov chain where

$$M_k[i_{t-k}, i_{t-k+1}, \ldots, i_t] = P(i_t | i_{t-k}, \ldots, i_{t-1}) \tag{5.2}$$

In the rest of this chapter, we denote $\underbrace{N \times N \times, \ldots, \times N}_{\text{d N's}}$ as $N^{(d)}$ for simplicity. Note that the coordinate $[i_{t-k}, i_{t-k+1}, \ldots, i_t]$ of an element in M_k inherently forms a path $\mathcal{I}_k^t$. The main goal of our proposed method is to effectively and efficiently reduce the candidate set for predictive range queries with the aid of (high order) Markov chains.

5.1.4 Sparse matrix storage

Transition matrices, especially those for high-order Markov chains, are usually sparse in spatio-temporal settings (see Section 5.3.2). We review some classic storage formats for sparse matrices [17, 39]. The *dictionary of keys* (DOK) format consists of a dictionary that maps (row, column)-pairs to values for non-zero elements. The DOK format is ideal for incrementally constructing the matrices but poor for arithmetic operations. The *compressed row storage* (CSR) format stores a sparse matrix using three 1-dimensional arrays (A, IA, JA), where A holds all the nonzero entries in *row-major* order, IA records the start and end indexes in A for each row, finally, JA contains the column index of each element of A. The CSR format is efficient for arithmetic operations such as inner-product, but inefficient for incremental construction. Therefore, one typical strategy is to use DOK format for construction and then convert the

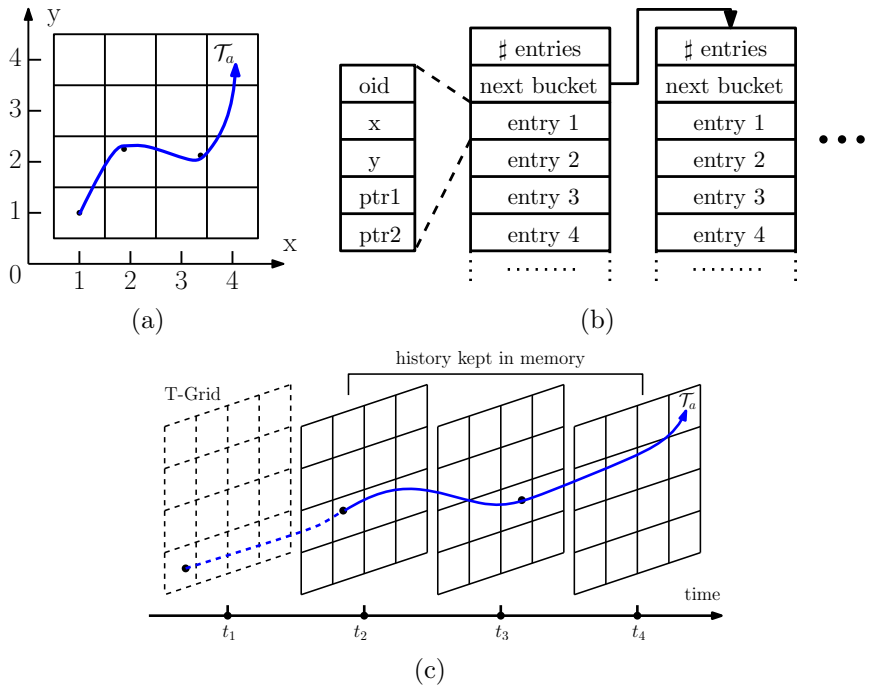


Figure 5.1: Structure of the T-Grid

matrix to CSR format while performing arithmetic operations. For matrices that consist of a few diagonals, the diagonal format (DIA) stores the diagonals in a rectangular $N_{diag} \times N$ array, where N_{diag} is the number of diagonals and N is the number of columns. Note that these formats only support 2-dimensional matrices but are not applicable in high dimensional scenarios. We present new data structures for storing the sparse high-dimensional transition matrices in next section.

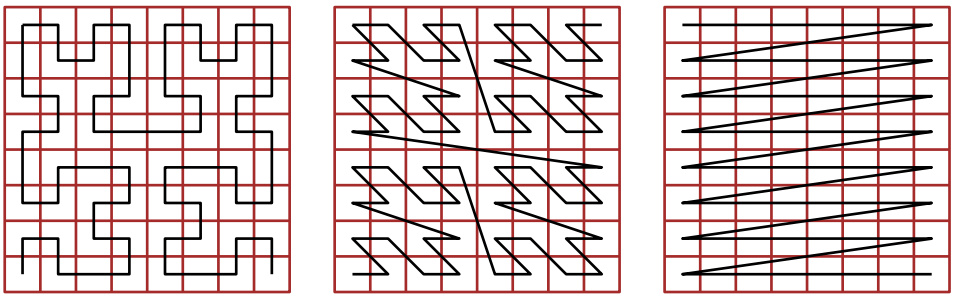
5.2 Data Structures

In this section, we introduce the data structures to store the object trajectories and the sparse Markov transition matrices.

5.2.1 The trajectory grid

Figure 5.1(a) shows part of a trajectory in the 2-dimensional uniform grid, where the dots represent the sampled locations. Note that we assume that all trajectories are sampled with equal intervals and synchronized timestamps. Interpolation is performed when this assumption does not hold. The *trajectory grid* (T-Grid) consists of a series of uniform grids aligned by timestamps, which we call *phases*. Each phase stores only the sampled locations with a certain timestamp. Figure 5.1(c) shows an example of the T-Grid and the trajectory that is identical to the one in Figure 5.1(a). Note that only the most recent H phases are kept in main memory and $H \geqslant k$, where k is the order of the underlying Markov chain. The main purpose of introducing phases is to support fast object/trajectory retrievals by timestamps, which is useful in query processing (see Section 5.3).

Sampled locations of the trajectories are stored in the corresponding grid cells. A grid cell is stored as a list of *buckets* where each bucket contains the number of data entries stored, the pointer that links to the next bucket when the number of entries in the current bucket exceeds a predefined *capacity*, and an array of data entries. Each data entry stores a single trajectory record, which consists of object ID (*oid*), spatial coordinate (x and y) and the pointer that links to the position of the next record in the same trajectory. Each such pointer contains two fields $ptr1$ and $ptr2$ that represent cell ID and in-bucket position of the next record (data entry), respectively. Figure 5.1(b) illustrates the storage structure of a grid cell. In this chapter, the capacity is 1000 by default.



(a) Hilber (b) Morton (c) Raster

Figure 5.2: Space-filling curves

5.2.2 The MDIA format

As mentioned in Section 5.1, we use Markov chains to encode consecutive states of the moving objects. In order to store the Markov transition matrices as a multi-dimensional arrays, we need to arrange the grid cells in 2-dimensional space with a 1-dimensional sequence via *space-filling*. We explored three different shapes of space-filling curves: Hilbert, Morton, and Raster, which are illustrated in Figure 5.2. The Markov transition matrices are usually extremely sparse in spatio-temporal settings, especially those of high order Markov chains. The reason is that, in real world scenarios, due to geographic restrictions and velocity limitations, any moving object, for example vehicles or pedestrians, can only move to nearby places rather than the entire predefined space domain within one sampling interval.

Table 5.2 shows occupancy ratios of the Markov transition matrices derived from the taxi data of Beijing city (see Section 5.4). In Table 5.2, the *Element* column represents the *element occupancy* while the *Hilbert*, *Morton*, and *Raster* columns represent the *diagonal occupancy* with the corresponding space-filling techniques. Element occupancy and diagonal occupancy denote the ratio of nonzero elements and that of (partially) occupied diagonals (see Definition 5.3),

Table 5.2: Occupancy ratio of Markov transition matrices

Grid size	Order	Element	Hilbert	Morton	Raster
64 × 64	order-1	0.003	0.175	0.093	**0.008**
	order-2	6.39e-10	3.89e-4	2.563e-4	**2.55e-5**
	order-3	5.56e-24	1.89e-7	1.308e-7	**2.17e-8**
128 × 128	order-1	8.59e-4	0.133	0.074	**0.007**
	order-2	1.1e-11	9.04e-5	5.7e-5	**7.72e-6**
	order-3	3.8e-28	1.36e-8	9.13e-9	**1.33e-9**
256 × 256	order-1	2.23e-4	0.096	0.054	**6.15e-4**
	order-2	1.79e-13	2.31e-5	1.41e-5	**1.81e-6**
	order-3	2.06e-32	9.29e-10	6.52e-10	**8.72e-11**

respectively. We find that the transition matrices become more and more sparse when either the order of Markov chain or the grid size increases. The transition matrices enjoy very low diagonal occupancy, which means that they can be fully and compactly represented by only a small portion of diagonals. The intuition is that moving objects transit to neighboring places, therefore non-zero transition probabilities tend to be clustered around diagonals. Moreover, among the three space-filling techniques, Raster gains the lowest diagonal occupancy, thus is selected as the default space-filling technique in this chapter. Additionally, for high-order Markov transition matrices, element occupancies are much lower than diagonal occupancies, which means that the diagonals themselves might be sparse as well.

Based on the above observations, we propose a multi-dimensional diagonal (MDIA) representation for extremely sparse matrices. The MDIA format of a sparse matrix consists of two components: 1) the diagonal matrix, $\mathcal{D}$, which is an $N_{diag} \times N$ matrix (using its CSR format [39] when sparse), where N_{diag} denotes the number of (major) diagonals, and 2) *offsets* (with respect to the major diagonal starting from the origin) of the diagonals, which form an array of tuples denoted as Ξ. We generalize the definition of diagonals and offsets

from 2-dimensional matrices [17, 39] to multi-dimensional matrices as follows.

Definition 5.3 (Diagonal and offset)**.** Given an $N^{(m)}$ ($m \geqslant 2$) matrix M, the tuple $\phi = (\xi, \mathbf{d})$ is called a diagonal of M if and only if $\mathbf{d}$ is a 1-dimensional array with $\mathbf{d}[i] = M[i + \delta_0, i + \delta_1, \ldots, i + \delta_{m-2}, i]$, where $0 \leqslant i + \delta_j < N, \forall 0 \leqslant i < N, 0 \leqslant j < m$ and $\xi = (\delta_0, \ldots, \delta_{m-2})$ is called the offset of ϕ.

According to this definition, the element $\mathbf{d}[i]$ is valid only when $0 \leqslant i+\delta_j < N$, $\forall 0 \leqslant j < m$, which is equivalent to $max(0, -\delta_{min}) \leqslant i < min(N, N-\delta_{max})$, where δ_{min} and δ_{max} denote the minimum and maximum values of δ_i, $0 \leqslant i \leqslant m-2$. We denote $\omega_\phi = [max(0, -\delta_{min}), min(N, N - \delta_{max}))$ as the *valid range* of ϕ. Given a matrix coordinate $\mathcal{I} =< i_0, i_1, \ldots, i_{m-1} >$, offset of the diagonal where it resides and the in-diagonal position are computed by

$$\xi = (i_0 - i_{m-1}, i_1 - i_{m-1}, \ldots, i_{m-2} - i_{m-1}),\ \kappa = i_{m-1} \tag{5.3}$$

On the contrary, given the offset $\xi = (\delta_0, \ldots, \delta_{m-2})$ and the in-diagonal position $\kappa \in \omega_\phi$, the corresponding coordinate is calculated by

$$\mathcal{I} =< \kappa + \delta_0, \kappa + \delta_1, \ldots, \kappa + \delta_{m-2}, \kappa > \tag{5.4}$$

We also define, for any 1-dimensional matrix/array M, $\mathbf{d} = M$ as the only diagonal with offset $\xi = \varnothing$.

Figure 5.3 shows examples of diagonals, $\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{d}_3$, and their offsets of a 2-dimensional matrix, where the solid part of $\mathbf{d}_1$ through $\mathbf{d}_3$ represent the valid ranges. The following shows an example of a high dimensional MDIA matrix.

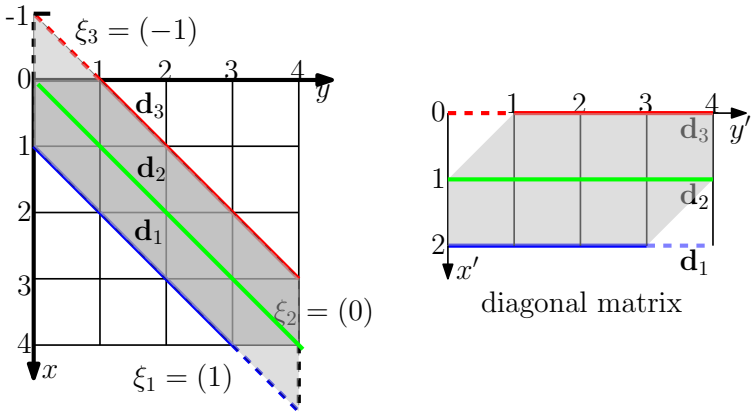
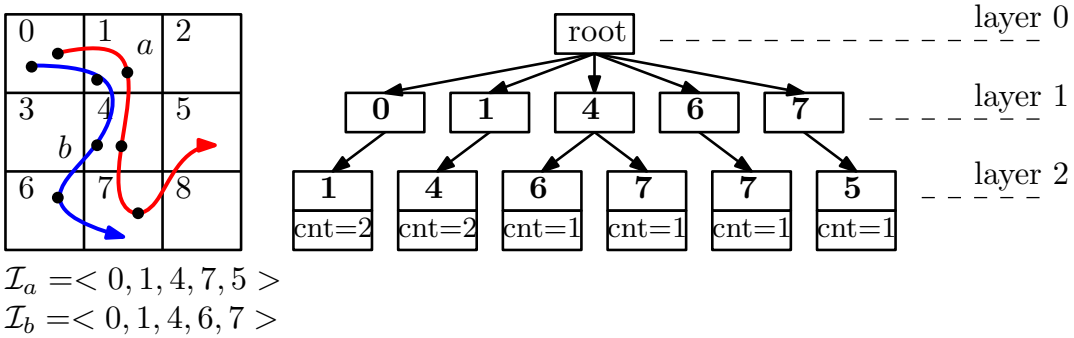


Figure 5.3: Diagonals

Example 5.1. Let M be a 3-dimensional matrix where $M[0,:,:] = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, $M[1,:,:] = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$, $M[2,:,:] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$. M has three diagonals $((0,0),[1,1,1])$, $((0,1),[1,1,0])$, and $((0,-1),[0,1,1])$. The MDIA format of M is $(\mathcal{D},\Xi)$ where $\Xi = \begin{bmatrix} (0,0) \\ (0,1) \\ (0,-1) \end{bmatrix}$ and $\mathcal{D} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$. The CSR format of $\mathcal{D}$ is (A, IA, JA), where $A = [1,1,1,1,1,1,1]$, $IA = [0,3,5,7]$, and $JA = [0,1,2,0,1,1,2]$.

5.2.3 The transition trie

In dynamic environments where the transition patterns change over time and new trajectories come into existence, the transition matrices need to be updated periodically. Although the MDIA format stores multi-dimensional matrices compactly and is efficient for arithmetic operations (see Section 5.3), unfortu-



$\mathcal{I}_a =< 0, 1, 4, 7, 5 >$
$\mathcal{I}_b =< 0, 1, 4, 6, 7 >$

Figure 5.4: Construct transition trie from paths

nately, it is inefficient for incremental construction or update, especially when the diagonal matrix is in CSR format. The DOK format (see Section 5.1) with a dictionary that maps matrix coordinates to values can be used to store multi-dimensional matrices naively. However, such structure consumes large storage sizes for large matrices especially when the dimension is high. We propose the *transition trie* structure to store all the historical path information of the moving objects which can be used to construct and update the transition matrices. In a transition trie, a path from root to leaf, denoted as $\mathcal{I}$, consists of a sequence of states that is equivalent to a matrix coordinate and each leaf node contains a *value field* that stores the count of occurrences of the corresponding path. Figure 5.4 shows an example of transition trie with count information for length-2 (sub)paths from path $\mathcal{I}_a$ (red) and $\mathcal{I}_b$ (blue), which can be used to construct and update the order-1 Markov transition matrix. Implementations of the insertion and deletion operations in transition trie are inherited from the classic trie data structure [12] and are omitted here. Transition tries are stored in disk and loaded in memory for generating transition matrices.

Algorithm 5.1: Generate transition matrix

```
   input  : T (transition trie loaded from disk)
   output: M_k (MDIA transition matrix)
 1 D ← dictionary (key: offset, value: diagonal array);
   /* iterate paths and counts in leaf nodes                    */
 2 forall the (I, cnt) in T do
 3 |   ξ, κ ← offset and position of I;          /* Equation (5.3) */
 4 |   if ξ not in data.keys then
 5 |   |   D[ξ] ← array of N 0's;
 6 |   D[ξ][κ] ← cnt;
 7 Ξ ← D.keys;
 8 if D.values is sparse then
 9 |   D ← csr_matrix(D.values);
10 else
11 |   D ← 2d_array(D.values);
12 normalize(D);      /* normalize D on the first k dimensions */
13 M_k ← MDIA(D, Ξ);
14 return M_k;
```

5.3 Algorithms

In this section, we introduce our proposed pruning algorithm for predictive range queries using Markov chains.

5.3.1 Markov chain based pruning

Generate transition matrix

We first introduce the approach of generating the Markov transition matrix in MDIA format from the paths stored in the transition trie, as summarized in Algorithm 5.1. Note that this operation is processed off-line periodically. Algorithm 5.1 first creates a dictionary to store the diagonals with their offsets as keys (line 1) and then iterates through paths in the transition trie T, where $\mathcal{I}$ and *cnt* denote the path and count (lines 2-6). In each iteration, it finds

the offset and in-diagonal position relating to the path/coordinate $\mathcal{I}$ using Equation (5.3) and records *cnt* in the dictionary (lines 3-6). Then it serializes the key-set of D as the offset array Ξ and sets the diagonal matrix $\mathcal{D}$, which is derived from the value-set of D. D is stored as a CSR matrix [39] if it is sparse otherwise as an ordinary matrix (a 2-dimensional array) (lines 7-11). Finally, $\mathcal{D}$ is normalized on the first k dimensions (line 12), so that it contains the transition probabilities that can be used to find the candidate paths.

Generate candidate paths

Given a predictive range query $Q(R, t_q)$, a binary vector $\mathbf{q}$ represents the spatial region R. Specifically, the i^{th} position in $\mathbf{q}$ is set when state s_i intersects with R. We call $\mathbf{q}$ the *query vector* of Q.

Definition 5.4 (State matrix)**.** Given a predictive range query $Q(R, t_q)$ with query vector $\mathbf{q}$, at any time t and $t_c \leqslant t \leqslant t_q$, a state matrix ρ_k^t of an order-k Markov chain is an $N^{(k)}$ matrix, where $\rho_k^t[i_{t-k+1}, \ldots, i_t] = P(\mathbf{q}|\mathcal{I}_k^t)$ denoting the probability that the k-backward path $\mathcal{I}_k^t =< i_{t-k+1}, \ldots, i_t >$ moves into R at time t_q.

In order to reduce the candidate set, we need to compute the probability for each object, given its base k-backward path, moving into R at time t_q. Such probabilities are stored in the state matrix $\rho_k^{t_c}$, which is called the *base state matrix*. Based on the local Markov property and Bayes rules, each element in

ρ_k^t, $t_c \leqslant t < t_q$, can be calculated by

$$\begin{aligned} P(\mathbf{q}|\mathcal{I}_k^t) &= \sum_{\forall i_{t+1} \in \mathcal{S}} P(\mathbf{q}|\mathcal{I}_k^{t+1}) P(\mathcal{I}_k^{t+1}|\mathcal{I}_k^t) \\ &= \sum_{\forall i_{t+1} \in \mathcal{S}} P(\mathbf{q}|\mathcal{I}_k^{t+1}) \frac{P(\mathcal{I}_{k+1}^{t+1})}{P(\mathcal{I}_k^t)} \\ &= \sum_{\forall i_{t+1} \in \mathcal{S}} P(\mathbf{q}|\mathcal{I}_k^{t+1}) P(i_{t+1}|\mathcal{I}_k^t) \end{aligned} \tag{5.5}$$

Note that $P(i_{t+1}|\mathcal{I}_k^t) \equiv P(i_{t+1}|i_{t-k+1}, i_{t-k+2}, \ldots, i_t)$ equals to $M_k[i_{t-k+1}, i_{t-k+1}, \ldots, i_{t+1}]$, where M_k is the order-k Markov transition matrix. Thus, we can rewrite Equation (5.5) in the following matrix form

$$\rho_k^t = \begin{cases} M_k \odot \mathbf{q} & t = t_q - 1 \\ M_k \odot \rho_k^{t+1}, & t_c \leqslant t < t_q - 1 \end{cases} \tag{5.6}$$

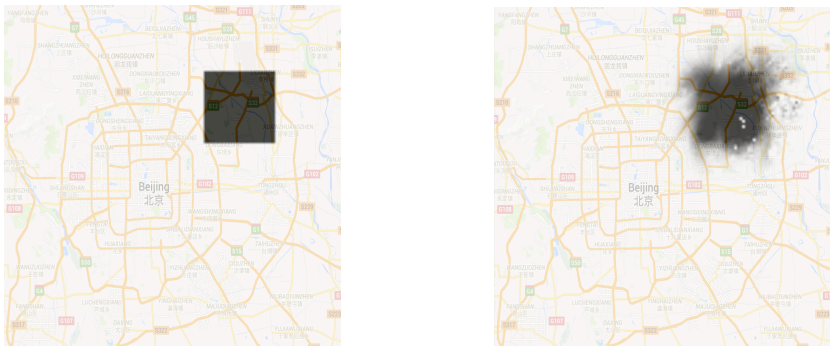
where ($\odot$) represents the *multiply* operation, which is defined as follows.

Definition 5.5 (Multiply)**.** Let M be an $N^{(m)}$ matrix, and L an $N^{(l)}$ matrix, where $m \geqslant 2$ and $1 \leqslant l < k$. $K = M \odot L$ is computed by

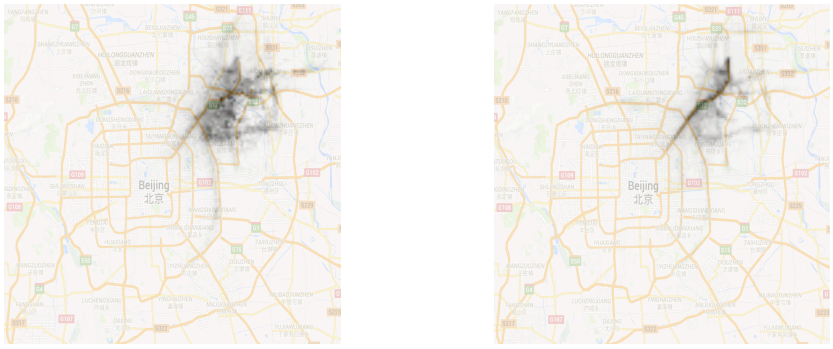
$$K[i_0, \ldots, i_{m-2}] = \sum_{0 \leqslant j < N} M[i_0, \ldots, i_{m-2}, j] L[i_{m-l}, \ldots, i_{m-2}, j] \tag{5.7}$$

Note that when $m = 2$, multiply degenerates to the classic inner-product operation between a 2-dimensional matrix and a 1-dimensional array. The base state matrix $\rho_k^{t_c}$ is computed by iteratively performing Equation (5.6).

Figure 5.5 visualizes the base state matrices computed with order-k ($1 \leqslant k \leqslant 3$) Markov transition matrices learned from the Beijing dataset (see Section 5.4). Grayscales of the masks in Figure 5.5(b) through 5.5(d) reflect values of



(a) Query window (b) Order-1



(c) Order-2 (d) Order-3

Figure 5.5: Base state matrices with different order of Markov chains

the corresponding elements in the base state matrices (probabilities of transiting into R at time t_q). We can see that the masks for high order Markov chains largely follow the underlying road network, which also implies the transition patterns of the vehicles. Intuitively, we prune the paths with lower probabilities (lighter grayscales) to obtain the candidate paths.

The candidate paths are stored in a transition trie (with empty value fields), which we call the *candidate path trie* (CPT). The CPT contains paths/coordinates corresponding to the nonzero elements in the base state matrix with values greater than a threshold ϵ, which we call the *pruning sensitivity*. Intuitively, the CPT stores base k-backward paths that are promising to transit into R at time t_q according to the base state matrix. Algorithm 5.2 summarizes our approach for computing the CPT. Given a query Q, an order-k Markov tran-

Algorithm 5.2: Generate candidate path trie

```
   input  : M_k (Markov transition matrix)
            Q (predictive range query)
            ε (pruning sensitivity)
   output: T (candidate path trie)
   /* compute base state matrix                                  */
 1 q ← query vector of Q;
 2 ρ_k^{t_q−1} ← M_k ⊙ q;                          /* Equation (5.6) */
 3 for t ← t_q − 2 to t_c do
 4     ρ_k^t ← M_k ⊙ ρ^{t+1};                      /* Equation (5.6) */
 5 T ← empty transition trie;
   /* iterate nonzero elements of ρ_k^{t_c}                      */
 6 foreach diagonal φ(ξ, d) in ρ_k^{t_c} do
 7     foreach nonezero element e in d do
 8         κ ← position of e within d;
 9         I ← coordinate of (ξ, κ);                 /* Equation (5.4) */
10         if M[I]) > ε then
11             T.insert(I);
12 return T;
```

sition matrix M_k, and the pruning sensitivity ϵ, this algorithm first computes the base state matrix $\rho_k^{t_c}$ using Equation (5.6) (lines 1-4). We will explain the implementation details for the multiply operation in the next subsection. Then it generates the coordinate for each nonzero element in $\rho_k^{t_c}$ according to Equation (5.4) (lines 8, 9) and inserts it into the CPT P if its value is greater than ϵ (lines 6, 7). We will empirically study the pruning effect with different sensitivity levels in Section 5.4. Note that Algorithm 5.2 is generic and applies for both ordinary and CSR diagonal matrices, which will be instantiated with different iterating approaches over the nonzero elements (lines 6, 7). The following shows an example of computing the puring predicate with an order-1 Markov transition matrix.

Example 5.2. Given three historical paths, $< 0, 1, 1 >$, $< 2, 1, 1 >$, and

Algorithm 5.3: Process predictive range query

```
input  : TG (T-Grid)
         Q (predictive range query)
         M_k (transition matrix)
output: R (query result)
1 P ← compute candidate path trie ;                    /* Algorithm 5.2 */
2 C ← from phase t_c of TG get objects in T's layer-1 states;
3 foreach object o in C do
4     I ← from TG get o's base k-backward path;
5     if I in P then
6         if PREDICT(o, Q) is true then
7             R.add(o);
8 return R;
```

$< 2, 0, 2 >$, the order-1 Markov transition matrix learned from these paths is $M_1 = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 0 \end{bmatrix}$. Given a query Q with query vector $\mathbf{q} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ and prediction time $t_q = 2$ (note that current time $t_c = 0$). The state matrices are computed by

$$\rho_1^1 = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 0 \end{bmatrix} \odot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix} \tag{5.8}$$

$$\rho_1^0 = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 0 \end{bmatrix} \odot \begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0.25 \end{bmatrix} \tag{5.9}$$

If $\epsilon = 0.2$, the candidate path trie is $\boxed{root} \longrightarrow \boxed{2}$, which means only objects in state 2 at the current time are considered in the verification step.

Process predictive range query

Now we discuss the query processing employing the CPT, as summarized in Algorithm 5.3. This algorithm first generates the CPT P using Algorithm 5.2 (line 1). Then it constructs a set $\mathcal{C}$ that consists of objects located in layer-1 states in P (see Figure 5.4) at time t_c (line 2). This is accomplished by retrieving objects from the latest phase of the T-Grid TG. Then each object $o \in \mathcal{C}$ is added to the result set if its base k-backward path is contained in P and it passes the $PREDICT$ test, which can be any prediction function mentioned in Section 1 (lines 3-7).

5.3.2 Discussions

Pre-computing

Alternative to Equation (5.6), the base transition matrix can also be obtained as follows

$$\rho_k^{t_c} = M_k^{t_q - t_c} \odot \mathbf{q} \tag{5.10}$$

where M_k^n is power n of M_k and denotes the n-step transition matrix, which records the probabilities of any base k-backward path $< i_0, \ldots, i_{k-1} >$ transiting to i_k and is computed by

$$\begin{aligned} &M_k^{n+1}[i_0, \ldots, i_{k-1}, i_k] = \\ &\sum_{0 \leqslant j < N} M_k[i_0, \ldots, i_{k-1}, j] M_k^n[i_1, \ldots, i_{k-1}, j, i_k] \end{aligned} \tag{5.11}$$

The powers of M_k can be pre-computed and, at runtime, only one multiply operation (Equation (5.10)) needs to be performed to obtain the CPT. However, the density (with respect to both element and diagonal occupancies) of M_k^n

grows as n increases. The MDIA format will eventually lose effect when n is large. On the other hand, the first k dimensions of M_k^n are always sparse, since the computation in Equation (5.11) only affects the very last dimension of M_k^n. Such a pattern can also be utilized to devise other formats instead of MDIA to effectively represent M_k^n. Studying the sparse pattern as well as the convergence property [27] of M_k^n are left as our future work.

Limitation of Markov chains

An order-k Markov chain assumes that the probability of transiting to the next state is determined by k most recent states. Generally speaking, transition probabilities of low order Markov chains are usually biased for predicting future states of the objects in real world scenarios. On the other hand, training a high order Markov chain requires more data, otherwise reliability of the transition probabilities will be affected. Moreover, the computation cost grows as the order of Markov chain increases (see Section 5.4). Therefore, the Markov chain model is not appropriate as a prediction function in the verification step. Other Markov models such as hidden Markov model [60] and hidden semi-Markov model [1] are better choices, where observations for hidden states are involved. However, Markov chains are effective for pruning i.e. reducing the candidate sets for predictive range queries. Consider Example 5.2. The base state matrix computed from order-1 Markov chain is $\begin{bmatrix} 0 \\ 0 \\ 0.25 \end{bmatrix}$, while the correct answer should be $\begin{bmatrix} 0 \\ 0 \\ 0.5 \end{bmatrix}$ (two of the three paths start from state 2 and one of

them enters state 2 at time 2). Although the order-1 Markov chain generates an incorrect result, it figures out that only objects in state 2 are possible to enter state 2 at time 2, which successfully reduces the candidate set. Thus, in this paper, we explore the pruning capabilities of Markov chains and leave their predicting capabilities to future work.

5.3.3 Multiply with MDIA matrices

The major computational burden of computing the CPT resides in processing the multiply operations (lines 2-4 in Algorithm 5.2), which cannot be computed explicitly using Equation (5.7), since the matrices are stored in MDIA format. The traditional inner-product operation for 2-dimensional matrices is performed with row or column-major order, i.e. sequentially compute each row or column of the result matrix. For example, while computing $K = M \odot L$, where M and L both are $N \times N$ ordinary matrices, the j^{th} column of K is calculated by performing inner-products between the j^{th} column of L and each row of M. However, this strategy does not apply to MDIA matrices, since the elements are arranged in diagonals instead of rows or columns. In this paper, we propose an algorithm that efficiently performs the multiply operation between MDIA matrices by using a diagonal-major strategy that computes diagonals of the result matrix in a predefined order.

Projection table

We first introduce the *projection table*, which links diagonals with their projections defined as follows. The major usage of projection table is to help us find the related diagonals during the multiply operation.

Definition 5.6 (Projection)**.** Let $\phi = (\xi, \mathbf{d})$ be a diagonal of an $N^{(m)}$ matrix,

where $m \geqslant 2$ and $\xi = (\delta_0, \delta_1, \ldots, \delta_{m-2})$. $\phi^{\perp} = (\xi^{\perp}, \mathbf{d}^{\perp})$ is an H-projection of ϕ if

$$\xi^{\perp} = \begin{cases} \varnothing & m = 2 \\ (\delta_0^{\perp}, \delta_1^{\perp}, \ldots, \delta_{m-3}^{\perp}) & m > 2 \end{cases} \tag{5.12}$$

where $\delta_i^{\perp} = \delta_i - \delta_{m-2}, \forall 0 \leqslant i < m-1$ and

$$\mathbf{d}^{\perp}[i + \iota] = \mathbf{d}[i], \forall 0 \leqslant i, i + \iota < N \tag{5.13}$$

where

$$\iota = \begin{cases} \delta_{m-2} & m = 2 \\ \delta_{m-2} - \delta_{m-3} & m > 2 \end{cases} \tag{5.14}$$

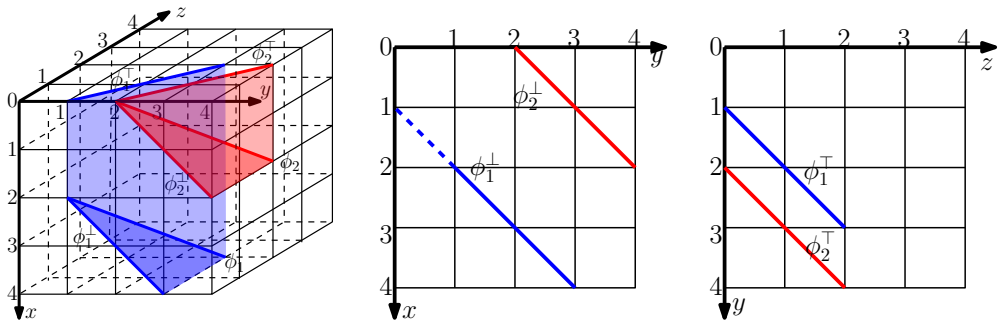
is the *shift* of the projection. $\phi^{\top} = (\xi^{\top}, \mathbf{d}^{\top})$ is a T-projection of ϕ if

$$\xi^{\top} = \begin{cases} \varnothing & m = 2 \\ (\delta_0^{\top}, \delta_1^{\top}, \ldots, \delta_{m-3}^{\top}) & m > 2 \end{cases} \tag{5.15}$$

where $\delta_i^{\top} = \delta_{i+1}$, $0 \leqslant i < m-1$ and

$$\mathbf{d}^{\top}[i] = \mathbf{d}[i], \forall 0 \leqslant i < N. \tag{5.16}$$

$(\perp)$ and $(\top)$ are called the H and T-projectors, respectively. Intuitively, the H and T-projections project a diagonal of an m-dimensional matrix onto the first and last $m-1$ dimensions, respectively. The projections themselves are partial diagonals of a $(m-1)$-dimensional matrix. Note that the array elements of ϕ are not vertically projected onto its H-projection $\phi^{\perp}$, but are aligned according to a shift computed by Equation (5.14). Figure 5.6(a) shows an example of projecting two diagonals ϕ_1 (blue) and ϕ_2 (red). The darker and



(a) (b) (c)

Figure 5.6: Projections

Algorithm 5.4: Create projection table

input : M (an $N^{(k+1)}$ MDIA matrix)
output: PT (projection table)

$PT, \Xi^\perp, \Xi^\top \leftarrow$ empty arrays;
foreach *diagonal* ϕ **in** M **do**
  `/* compute projections of` ϕ, $\phi^\perp = (\xi^\perp, \mathbf{d}^\perp)$, $\phi^\top = (\xi^\top, \mathbf{d}^\top)$ `*/`
  $\phi^\perp, \phi^\top \leftarrow H$ and T-projections of ϕ;
  if $\xi^\perp$ **not in** $\Xi^\perp$ **then**
    $\Xi^\perp.append(\xi^\perp)$;
  if $\xi^\top$ **not in** $\Xi^\top$ **then**
    $\Xi^\top.append(\xi^\top)$;
  $\alpha, \beta, \gamma \leftarrow$ index of $\xi, \xi^\perp, \xi^\top$ in Ξ, $\Xi^\perp$, $\Xi^\top$;
  $PT.append(\alpha, \beta, \gamma)$;
`/* create hash index */`
create index IDX_β on $PT(\beta)$;
return PT;

lighter areas denote the sweeping regions of projecting the diagonals onto the xy and yz sub-spaces, respectively. Figure 5.6(b) and 5.6(c) show the H and T-projections in the corresponding sub-spaces, respectively. The dashed line in Figure 5.6(b) illustrates the shift over the H-projection of ϕ_1.

The projection table associated with an MDIA matrix $M(\mathcal{D}, \Xi)$ is obtained by performing H and T-projections on every single diagonal of M and collecting **distinct** offsets of the projections, denoted as $\Xi^\perp$ and $\Xi^\top$. The projection table

is a 2-dimensional array that stores one tuple (α, β, γ) in each line, where α, β, and γ represent the index of ξ, $\xi^{\perp}$, and $\xi^{\top}$, in Ξ, $\Xi^{\perp}$, and $\Xi^{\top}$, respectively. The following shows an example of computing projections and generating the projection table.

Example 5.3. Let $M(\mathcal{D}, \Xi)$ be an MDIA matrix, where

$$\mathcal{D} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \text{ and } \Xi = \begin{bmatrix} (0,0) \\ (1,1) \\ (0,-1) \end{bmatrix}$$

The H-projections are

$$\mathcal{D}^{\perp} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \text{ and } \Xi^{\perp} = \begin{bmatrix} (0) \\ (1) \end{bmatrix}$$

while the T-projections are

$$\mathcal{D}^{\top} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \text{ and } \Xi^{\top} = \begin{bmatrix} (0) \\ (1) \\ (-1) \end{bmatrix}$$

Finally, the projection table is

$$PT = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 2 \end{bmatrix}$$

The main purpose of introducing the projection table is to instantly locate the related diagonals during the multiply operation. For instance, when we compute $K = M \odot L$, the projection table should help us find the corresponding diagonals of M and L that contain all the data that is needed to compute a certain diagonal of K. Thus, we create a hash index, IDX_{β} on the second

Algorithm 5.5: Multiply

input : M (an $N^{(k+1)}$ MDIA matrix)
L (an $N^{(k)}$ MDIA matrix)
PT (projection table)
output: K $(L \odot M)$

```
1  N_diag ← number of distinct β in PT;
2  Ξ ← array of N_diag tuples;
3  D ← N_diag × N array of 0's;
4  for β ← 0 to N_diag − 1 do
       /* get PT lines associated with β                          */
5      foreach (α, β, γ) in PT.IDX_β[β] do
6          for i ← 0 to N − 1 do
7              j ← i + ι;                          /* Equation (5.14) */
8              if 0 ⩽ j < N then
9                  D[β][j] ← D[β][j] + M.D[α][i] × L.D[γ][i];
10     Ξ[β] ← M.Ξ[α]^⊥ ;                           /* Equation (5.12) */
11 K ← MDIA(D, Ξ);
12 return K;
```

column of the projection table to support constant time retrievals by β. i.e. for each diagonal $\phi^{\perp}$ of K, we can quickly find the diagonals of M that project onto $\phi^{\perp}$ and their T-projections. Algorithm 5.4 summarizes the main steps of generating the projection table. Next we present the approach for multiplying Markov transition matrices and state matrices with the help of projection tables. Multiplication between Markov transition matrices and query vectors can be performed similarly, and is therefore omitted here.

Dense diagonal matrix

Algorithm 5.5 summarizes the approach to perform multiply operation when the diagonal matrix $\mathcal{D}$ is dense and stored as a 2-dimensional array. In Algorithm 5.5, we first allocate the memory for storing the MDIA format of K (lines 1-3). Then we compute the array $\mathbf{d}^{\perp} = \mathcal{D}[\beta]$ (lines 5-9) and offset

Algorithm 5.6: Multiply (CSR)

```
   input  : M (an N^(k+1) MDIA matrix)
            L (an N^(k) MDIA matrix)
            PT (projection table)
   output: K (L ⊙ M)
 1 N_diag ← number of distinct β in PT;
 2 Ξ ← array of N_diag tuples;
   /* pass one:  compute IA                                              */
 3 IA ← array of N_diag + 1 0's;
 4 mask ← array of N -1's;
 5 nnz ← 0;
 6 for β ← 0 to N_diag − 1 do
 7     foreach (α, β, γ) in PT.IDX_β[β] do
 8         ii ← M.IA[α], jj ← L.IA[γ];
 9         while ii < M.IA[α + 1] and jj < L.IA[γ + 1] do
10             x ← M.JA[ii], y ← L.JA[jj];
11             if x = y then
12                 z ← x + ι;
13                 if mask[z] ≠ α then
14                     mask[z] ← α, nnz ← nnz + 1;
15                 ii ← ii + 1, jj ← jj + 1;
16             else if x > y then
17                 jj ← jj + 1;
18             else ii ← ii + 1;
19     IA[α + 1] ← nnz;
   /* pass two:  compute A and JA                                        */
20 A ← array of nnz 0's, JA ← array of nnz 0's;
21 next ← array of N 0's, sums ← array of N -1's;
22 nnz ← 0;
23 for β ← 0 to N_diag − 1 do
24     head ← -2, len ← 0;
25     foreach (α, β, γ) in PT.IDX_β[β] do
26         ii ← M.IA[α], jj ← L.IA[γ];
27         while ii < M.IA[α + 1] and jj < L.IA[γ + 1] do
28             x ← M.JA[ii], y ← L.JA[jj];
29             if x = y then
30                 z ← x + ι;
31                 sums[z] ← sums[z] + M.A[ii] × L.A[jj]; if next[z] = −1 then
32                     next[z] ← head, head ← z;
33                     len ← len + 1;
34                 ii ← ii + 1, jj ← jj + 1;
35             else if x > y then
36                 jj ← jj + 1;
37             else ii ← ii + 1;
38     for j ← 0 to len do
39         A[nnz] ← sums[head], JA[nnz] ← head;
40         nnz ← nnz + 1;
41         tmp ← head, head ← next[head];
42         next[tmp] ← -1, sums[tmp] ← 0;
43     Ξ[β] ← M.Ξ[α]^⊥
44 D ← csr_matrix(A, IA, JA);
45 sort_index(D);                          /* sort diagonals in ascending order */
46 return MDIA(D, Ξ);
```

$\xi^{\perp} = M.\Xi[\alpha]^{\perp}$ (line 10), projected from diagonals of M, for each diagonal of K. Finally we create the MDIA format of K from $\mathcal{D}$ and Ξ and return the

result (lines 11, 12).

Sparse diagonal matrix. As we mentioned in Section 5.2.2, the diagonal matrix $\mathcal{D}$ of a Markov transition matrix might be sparse, in which scenario, it is stored in CSR format. We also propose an approach, summarized in Algorithm 5.6, to process the multiply operation between CSR diagonal matrices, i.e. $\mathcal{D}$ is represented by three 1-dimensional arrays A, IA, and JA (see Section 5.1). Specifically, Algorithm 5.6 employs a *two-pass* approach. In the first pass (lines 3-19), it computes IA, which indicates the number of nonzero elements in each diagonal of K; then in the second pass (lines 20-43) it fills A and JA with the values and in-diagonal positions of the nonzero elements of K. Finally, we sort the element indexes within each diagonal in ascending order using the *sort_index*() method (line 44). The sorting is an important procedure that keeps the MDIA format valid during the multiply operation. The implementation of *sort_index*() is explained in [35, 39] and omitted here.

Complexity analysis

Given an $N^{(k)}$ matrix, if no sparse storage format is applied, both space and time complexity for processing the multiply operation is $O(N^k)$. For example, consider a 100×100 uniform grid, thus $N = 10,000$. M_2 is a $10,000 \times 10,000 \times 10,000$ matrix which contains 10^{12} elements. If each element is a 4-byte float number, M_2 will require 3 terabytes memory. Processing a single multiply operation will take hundreds of minutes on a 1GHz CPU. If M is in the MDIA format with ordinary diagonal matrix, the space and time complexities reduce to $O(N_{diag} \cdot N)$, where $N_{diag} \ll N^{k-1}$ denotes the number of diagonals in M. The diagonal matrix is stored in CSR format if it is sparse, i.e. the number of nonzero elements $NNZ \ll N \times N_{diag}$, in which scenario, the space and

Table 5.3: Experimental settings

Space domain (m×m)	**50,000×50,000**
Number of trajectories	**50,000**
Grid size	32 × 32, 64 × 64, **128 × 128**, 256 × 256
Order of Markov chain	1, **2**, 3, 4
Sampling interval (s)	30, **60**, 90, 120
Pruning sensitivity	10^{-6}, 10^{-5}, ..., 0,1, **0.2**, ..., 0.5
Window length (km)	1, **2**, 3, ..., 10
Query predict length (min)	10, **20**, 30, ..., 60
Dataset	**Beijing**, Shenzhen

time complexities further reduce to $O(NNZ + N_{diag})$, which is the theoretical optimum since any exact (in contrast to probabilistic) algorithm must iterate through every single diagonal and nonzero element to perform the multiply operations.

5.4 Experimental Study

In this section, we evaluated our proposed method and compare it with other existing methods. The experimental goal is to demonstrate that our method is effective and efficient in reducing the candidate set for predictive range queries. All algorithms are implemented with Python/C. The experimental machine is equipped with 2.6 GHz Intel Core i7 CPU and 16GB RAM and runs OSX 10.11.

5.4.1 Experimental setup

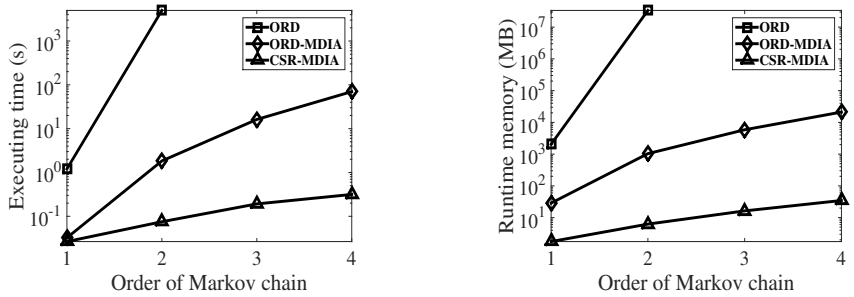
The experimental settings are shown in Table 5.3 where the default settings are boldfaced.

Datasets

We use two real world GPS tracking datasets: Beijing and Shenzhen. The Beijing dataset contains 15 million records collected from 10,358 taxis between Feb. 2, 2008 to Feb. 8, 2008 while the Shenzhen dataset contains 0.79 billion records collected from 26,572 taxis between March 20, 2014 to March 29, 2014. Each record includes the taxi ID, the location (longitude and latitude) and the timestamp. We select an 50,000 $\times$ 50,000 square meter area in the corresponding road networks as the space domain. We sample vehicle trajectories from the GPS records and perform interpolations to make all trajectories have equal sampling intervals and synchronized timestamps. From each dataset, we collect 50,000 one day long trajectories and divide them into two parts. The first part contains 40,000 trajectories that are used for training the models. The remaining 10,000 trajectories are prepared for testing. We sample 50,000 two hour long trajectories from the testing data and insert them into the T-Grids. Predictive range queries are randomly generated with the parameters specified in each experiment.

Evaluation metrics

We consider three metrics to evaluate the effect of our pruning methods: selectivity, precision and recall. Given the set of moving objects $\mathcal{O}$ with their trajectories and a predictive range query Q, selectivity equals $\frac{|\mathcal{O}'|}{|\mathcal{O}|}$, where $\mathcal{O}'$ is the candidate set and $|\cdot|$ denotes the cardinality of a set. Selectivity can be used to evaluate the pruning effect of a pruning method. Precision and recall are conventional measures of accuracy. Let $\mathcal{R}$ denote the result set of query Q. Precision equals $\frac{|\mathcal{O}' \cap \mathcal{R}|}{|\mathcal{O}'|}$ while recall equals $\frac{|\mathcal{O}' \cap \mathcal{R}|}{|\mathcal{R}|}$. In addition to these three metrics, we also evaluate the time and space costs of performing the pruning



(a) Time cost (b) Space cost

Figure 5.7: Varying order of Markov chain

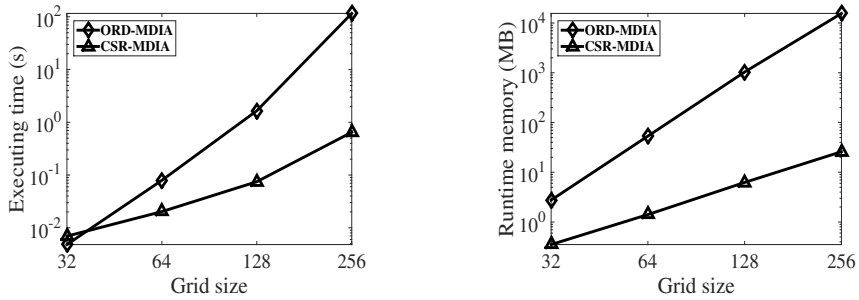
and query processing.

Competitors

We compared three alternative pruning methods with our proposed method. They are (1) Maximum speed (MS). This is a naive method that uses the maximum possible speed of all objects to enlarge the query window. (2) Velocity histogram (VH). This method is used in [18], which enlarges query windows by the maximum speed values of the objects in every grid cell that are stored in a 2-dimensional histogram. (3) Travel time grid (TG). This method is proposed in [15]. It tracks the average travel time between two grid cells, based on which the objects are not reachable to the query window within the prediction time are pruned.

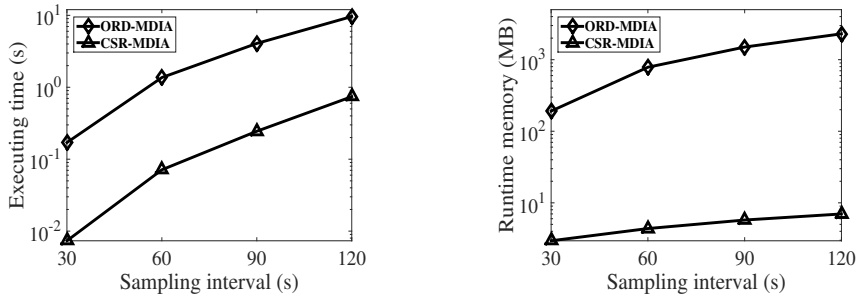
5.4.2 Evaluation of our methods

We first evaluate the performance of our proposed method. We evaluate time and space costs of computing the CPTs and the pruning effect of Markov chains. We compare the baseline ordinary matrices (ORD) with two variants of the proposed MDIA format, MDIA with ordinary diagonal matrices (ORD-



(a) Time cost (b) Space cost

Figure 5.8: Varying grid size



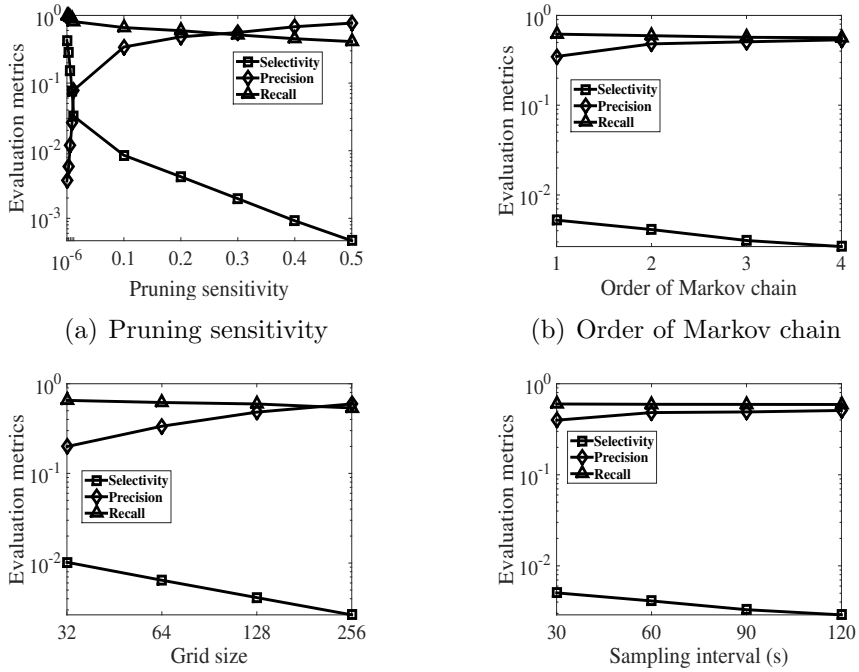
(a) Time cost (b) Space cost

Figure 5.9: Varying sampling interval

MDIA) and MDIA with CSR diagonal matrices (CSR-MDIA).

Time and space costs

Figure 5.7 shows time and space costs with varying order of Markov chains. We find that both execution time and memory rapidly grow as the order increases. Specifically, the size of Markov transition matrices grow exponentially, which makes ORD and ORD-MDIA infeasible with high-order Markov chains. However, CSR-MDIA is not as significantly affected by the order of Markov chains as other approaches. This is because the time and space complexity of CSR-MDIA is determined by the number of nonzero elements and that of occupied diagonals, which are far smaller than the size of the transition matrices.

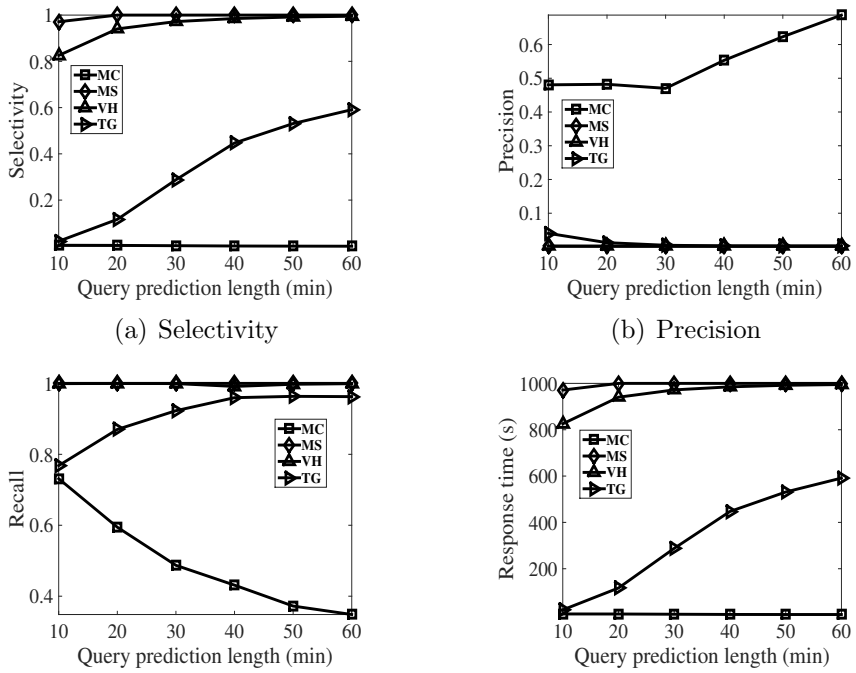


(a) Pruning sensitivity (b) Order of Markov chain

(c) Grid size (d) Sampling interval

Figure 5.10: Pruning effects with different parameters

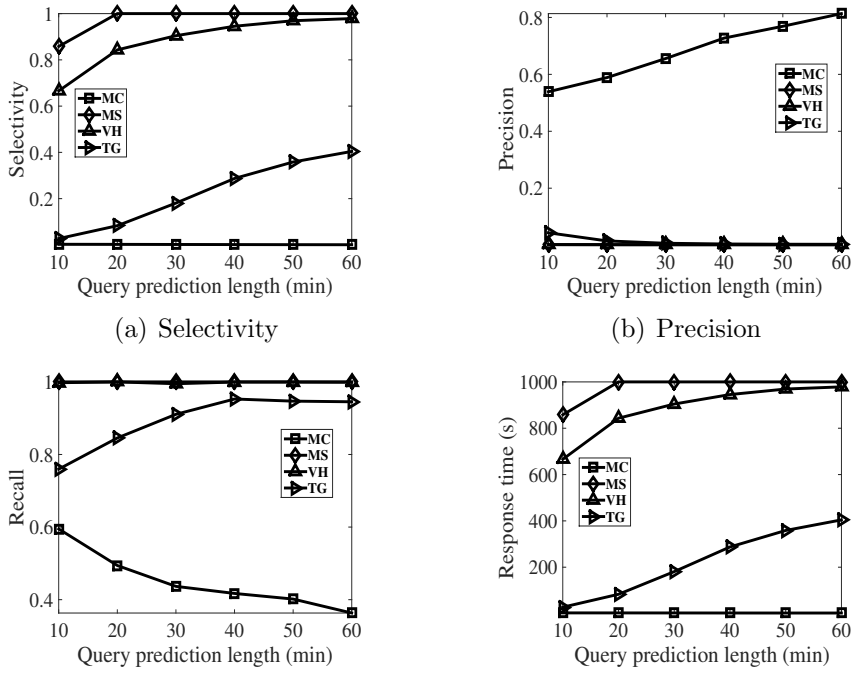
Figure 5.8 shows the effect of varying grid sizes. We omit ORD here since it is not viable with order-2 (or greater) Markov chains. As before, both time and space requirements grow as grid size increases, since larger grids result in more states and larger transition matrices. Moreover, CSR-MDIA significantly outperforms ORD-MDIA as grid size increases. Generally speaking, the order of Markov chain has more impact on the transition matrix than grid size, since growth in order increases the dimensions of the matrix instead of increasing the scope of each dimension. Figure 5.9 shows the impact of varying the sampling interval. We find that larger sampling intervals lead to higher computation costs, since the transition matrices get denser as the sampling interval increases. It is noteworthy that CSR-MDIA can accomplish the computation for CPTs within one second and ∼10 megabytes of memory in all experimental settings.



(a) Selectivity (b) Precision

(c) Recall (d) Response time

Figure 5.11: Varying query prediction length on Beijing dataset



(a) Selectivity (b) Precision

(c) Recall (d) Response time

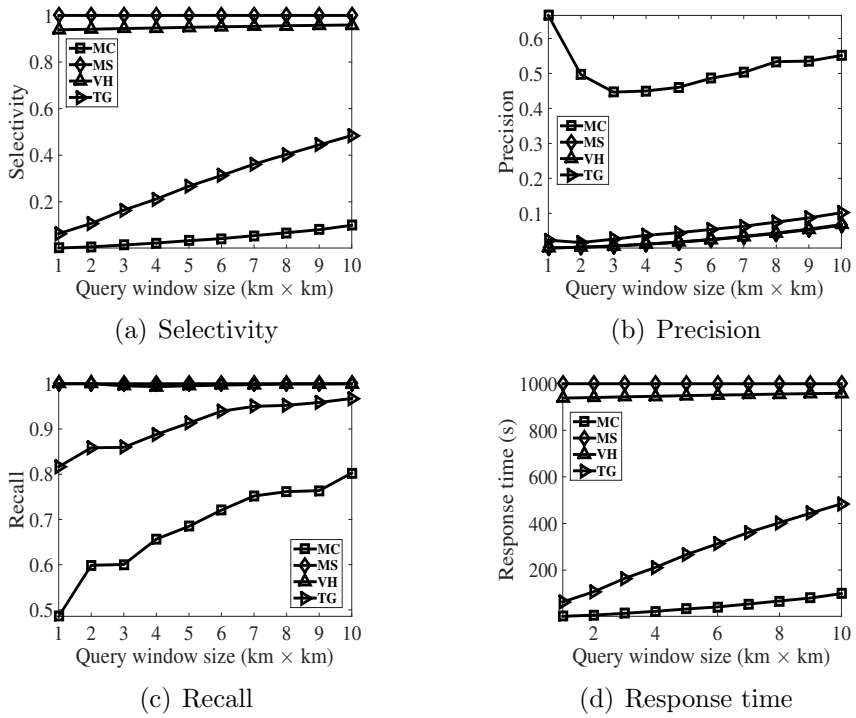
Figure 5.12: Varying query prediction length on Shenzhen dataset

Pruning effect

In this experiment, we evaluate the pruning capability of Markov chains by comparing selectivity, precision and recall. The experimental results are summarized in Figure 5.10. Figure 5.10(a) shows the impact of pruning sensitivity (threshold), by varying its value from 10^{-6} to 0.5. We find that selectivity significantly reduces when pruning sensitivity increases. Moreover, precision climbs while recall drops as pruning sensitivity increases. The reason is that the higher the pruning sensitivity, the more objects are likely to be filtered out, resulting in smaller candidate set thus lower selectivity. Figure 5.10(b) shows the results with different orders of Markov chain. We find that both selectivity and recall significantly reduce while precision rises as order increases. Therefore, high-order Markov chains are more powerful in pruning while scarifying prediction rate. Similarly, as shown in Figure 5.10(c), larger grid sizes (more finegrained grid cells) will result in fewer candidate paths, and thus result in lower selectivity and recall but higher precision. Similar trends with different sampling intervals are shown in Figure 5.10(d).

5.4.3 Comparison with other methods

In this set of experiments, we compare the performance of our proposed Markov chain based method (MC) with three existing methods, maximum speed (MS), velocity histogram (VH), and travel time grid (TG). We conduct experiments on both Beijing and Shenzhen datasets. Besides pruning effects, we also evaluate the overall query response time. We use the prediction approach introduced in [60] (a hidden Markov model based approach) as the $PREDICT()$ function in Algorithm 5.4.



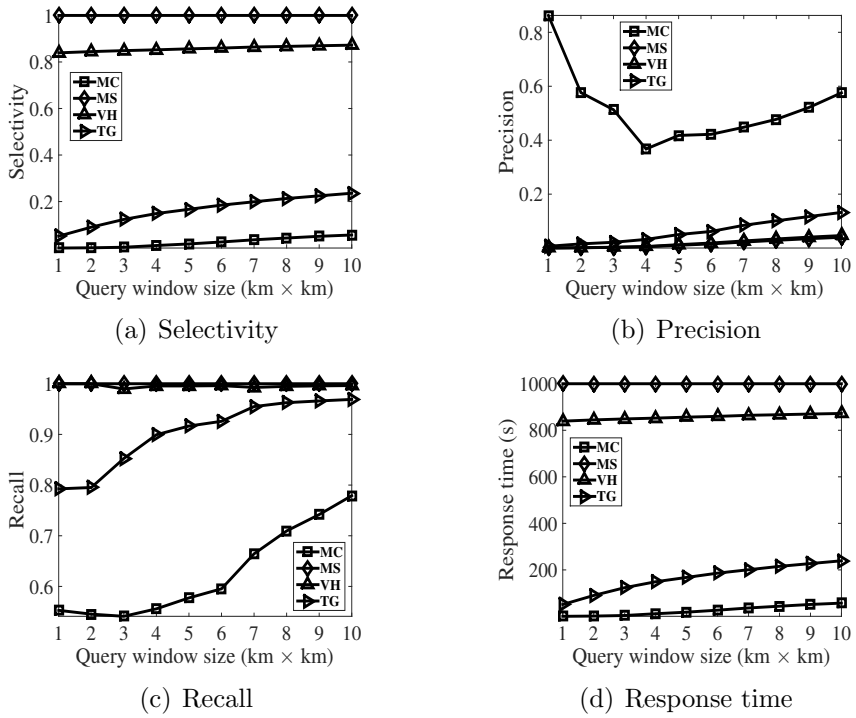
(a) Selectivity (b) Precision

(c) Recall (d) Response time

Figure 5.13: Varying query window size on Beijing dataset

Query prediction length

We first study the impact of query prediction length and vary its value from 10 to 60 minutes. Figure 5.11 and 5.12 show the results on Beijing and Shenzhen datasets, respectively. We find that selectivity and response time grow as prediction length increases. This is because uncertainty of the object trajectories increases for longer term prediction, thus we include a larger portion of candidate objects. Moreover, the response time is nearly linear with selectivity, which implies that performing the $PREDICT()$ function with candidate objects dominates the execution time for answering predictive range queries. Precision drops as prediction time increases, since longer-term predictions are more challenging. Recall of our method drops as prediction time increases while those for other methods do not change much. This is because the main goal



(a) Selectivity (b) Precision

(c) Recall (d) Response time

Figure 5.14: Varying query window size on Shenzhen dataset

of our method is to accurately and efficiently reduce the candidate set with a trade-off of prediction rate, while other methods have less effects on pruning. However, this trade-off can be adjusted by the pruning sensitivity. Similar trends are displayed for Beijing and Shenzhen datasets. We find our method works a little better on Shenzhen data. This is probably because vehicle speeds in Shenzhen city are higher than those in Beijing city, thus the motions can be better captured by the transition probabilities. In summary, our method significantly outperforms other methods in terms of pruning capability.

Query window size

We also evaluate performance of the methods with different query window sizes. Figure 5.13 and 5.14 show the results on Beijing and Shenzhen datasets, respectively. Generally speaking, the pruning procedure becomes more effective as

the query window increases. Similarly, we find that the response time is nearly linear to selectivity and our method works better on the Shenzhen dataset in different settings of query window sizes. Again, compared with other methods, our method enjoys significantly better performance in terms of selectivity, precision, and response time but scarifies recall.

Chapter 6

Conclusion and Future Work

6.1 Summary

In this dissertation, I present novel approaches for improving the efficiency of predictive spatio-temporal queries on large MODs.

In Chapter 3, I present the speed partitioning technique that aims to improve the query performances for tree-based indexes. We first formally defined the search space expansion which can be used as a generic metric to evaluate query performances for tree-based indexes on MODs. Via analyzing the impact of object velocities on the search space expansion, we proposed a novel and generic speed partitioning technique that significantly improves the query performances of tree-based indexes in terms of both range queries and k nearest neighbor queries. This method computes the optimal ranges for speed partitioning and an optional second-level partitioning over directions of the moving object velocities using dynamic programming. We applied the proposed speed partitioning technique on the state-of-the-art indexing structures including the B^x-tree and the TPR*-tree for experimental studies, which demonstrated that

our methods significantly outperforms the baseline approaches without considering velocity-based partitioning as well as other existing velocity-based partitioning techniques.

Chapter 4 continues the topic of using velocity information to improve query performance. Since the speed partitioning technique presented in Chapter 3 only applies on tree-based indexes, in this chapter, we explore the uniform grid structure, which has been proved more efficient for indexing MODs in main memory. We proposed the D-Grid, a novel dual space uniform grid that indexes MODs in the location-velocity dual space. The dual space is a $2d$-dimensional Euclidean space, where the first d dimensions represent velocity and the other d dimensions represent location. D-Grid leverages v-grids and l-grids in a manner that query window enlargements are significantly reduced in comparison with using location grids alone, thus is effective for improving query performance. We also proposed a lazy deletion and garbage cleaning (LDGC) mechanism which is effective in reducing update costs for uniform grid indexes, including D-Grid and other existing uniform grid indexes. Algorithms for both range queries and kNN queries on D-Grid are also discussed in this chapter.

Finally, in Chapter 5, we studied the usefulness of (high-order) Markov chains as a pruning mechanism for long-term predictive range queries. Since the verification step, which verifies validities of the objects with the query predicate, contributes to the major computational burden in typical pruning-verification strategy for processing predictive range queries, the more objects pruned in the pruning step, the less the query processing time. Motion functions, which are extensively used in performing short-term predictive range queries, are not suitable for long-term predictions since motions of the objects can change over time. Moreover, existing pruning techniques for long-term predictive range

queries suffer from limited pruning capacities. We proposed a (high-order) Markov chain based method that efficiently and effectively prunes the eearch space for long-term predictive range queries. In order to resolve the explosion of time and space costs arising with the computations of (high-order) Markov transition matrices, we propose the multi-dimensional diagonal (MDIA) format to compactly store the Markov transition matrices. The MDIA format stores only the partially occupied major diagonals of multi-dimensional matrices. This storage format is effective since non-zero probabilities in the transition matrices usually cluster around major diagonals in spatio-temporal settings. We also proposed efficient algorithms for arithmetic operations involved in our pruning procedure with MDIA matrices.

6.2 Future Work

To end this dissertation, I present some future works in related fields.

Velocity information is featured in spatio-temporal settings and can be utilized for performance improvement. Analytic methods such as *kernel density estimation* (KDE), instead of empirical methods, can be used to estimate the speed/velocity distributions, which helps find motion patterns of the moving objects. Velocity-based partitioning on the indexes has been proved effective in reducing query costs for spatio-temporal queries. An accurate estimation on the search space expansion can always help find a better partitioning mechanism. In dynamic scenarios, where the distributions of locations and speeds change frequently, sophisticated partition update algorithms will be necessary in maintaining the benefit of velocity-based partitioning on query performance. Moreover, hierarchical grids with predefined sizes might enhance the perfor-

mance of simple uniform grids in highly skewed datasets. Multi-threading can always boost both query and update operations with multiple CPUs or cores.

We have proposed an effective pruning mechanism that improves the performance for long-term predictive range queries via (high-order) Markov chains. The key of our method is storing the sparse Markov transition matrices in a compact manner. We will also seek efficient formats for representing the n-step Markov transition matrices according to their sparse patterns and further accelerate the multiply operations. Moreover, the convergence property of n-step transition probabilities is another interesting problem in spatio-temporal settings. Pre-computing popular queries according to the workload will further improve performance of the system. We are interested in discovering effective prediction functions using Markov chain models.

Finally, we will also consider privacy issues arisen with location sharing.

Bibliography

[1] M. Baratchi, N. Meratnia, P. J. M. Havinga, A. K. Skidmore, and B. A. G. Toxopeus. A hierarchical hidden semi-markov model for modeling mobility data. In *The 2014 ACM Conference on Ubiquitous Computing, UbiComp '14, Seattle, WA, USA, September 13-17, 2014*, pages 401–412, 2014.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 322–331, 1990.

[3] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *VLDB J.*, 15(3), 2006.

[4] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[5] P. Celis and J. V. Franco. The analysis of hashing with lazy deletions. *Inf. Sci.*, 62(1-2):13–26, 1992.

[6] S. Chen, C. S. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. *PVLDB*, 1(2):1574–1585, 2008.

[7] S. Chen, B. C. Ooi, K. Tan, and M. A. Nascimento. St^{2}b-tree: a self-tunable spatio-temporal b$^+$-tree index for moving objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 29–42, 2008.

[8] J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing moving objects using short-lived throwaway indexes. In *Advances in Spatial and Temporal Databases, 11th International Symposium, SSTD 2009*, pages 189–207, 2009.

[9] U. Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11, 2007.

[10] T. Emrich, H. Kriegel, N. Mamoulis, M. Renz, and A. Züfle. Querying uncertain spatio-temporal data. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 354–365, 2012.

[11] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[12] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9), 1960.

[13] J. Froehlich and J. Krumm. Route prediction from trip observations. Technical report, SAE Technical Paper, 2008.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57, 1984.

[15] A. M. Hendawi and M. F. Mokbel. Predictive spatio-temporal queries: a comprehensive survey and future directions. In *Proceedings of the First ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems, MobiGIS 2012, Redondo Beach, CA, USA, November 6, 2012*, pages 97–104, 2012.

[16] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee. Performance evaluation of main-memory r-tree variants. In *Advances in Spatial and Temporal Databases, 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 24-27, 2003, Proceedings*, pages 10–27, 2003.

[17] E.-J. Im and K. A. Yelick. *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[18] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 768–779, 2004.

[19] C. S. Jensen, H. Lu, and B. Yang. Indexing the trajectories of moving objects in symbolic indoor space. In *Advances in Spatial and Temporal Databases, 11th International Symposium, SSTD 2009, Aalborg, Denmark, July 8-10, 2009, Proceedings*, pages 208–227, 2009.

[20] H. Jeung, Q. Liu, H. T. Shen, and X. Zhou. A hybrid prediction model for moving objects. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 70–79, 2008.

[21] H. Jeung, M. L. Yiu, X. Zhou, and C. S. Jensen. Path prediction and predictive range querying in road network databases. *VLDB J.*, 2010.

[22] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 139–150, 2001.

[23] J. Krumm. Real time destination prediction based on efficient routes. Technical report, SAE Technical Paper, 2006.

[24] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *Proceedings of the Third International Conference on Mobile Data Management (MDM 2002), Singapore, January 8-11, 2002*, pages 113–120, 2002.

[25] M. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *VLDB*, pages 608–619, 2003.

[26] W. Mathew, R. Raposo, and B. Martins. Predicting future locations with hidden markov models. In *The 2012 ACM Conference on Ubiquitous Computing, Ubicomp '12, Pittsburgh, PA, USA, September 5-8, 2012*, pages 911–918, 2012.

[27] S. P. Meyn and R. L. Tweedie. *Markov chains and stochastic stability*. Springer Science & Business Media, 2012.

[28] A. Monreale, F. Pinelli, R. Trasarti, and F. Giannotti. Wherenext: a location predictor on trajectory pattern mining. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery*

and Data Mining, Paris, France, June 28 - July 1, 2009, pages 637–646, 2009.

[29] M. Morzy. Prediction of moving object location based on frequent trajectories. In *Computer and Information Sciences - ISCIS 2006, 21th International Symposium, Istanbul, Turkey, November 1-3, 2006, Proceedings*, pages 583–592, 2006.

[30] M. Morzy. Mining frequent trajectories of moving objects for location prediction. In *Machine Learning and Data Mining in Pattern Recognition, 5th International Conference, MLDM 2007, Leipzig, Germany, July 18-20, 2007, Proceedings*, pages 667–680, 2007.

[31] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 634–645, 2005.

[32] A. Nanopoulos, M. Vassilakopoulos, and Y. Manolopoulos. Performance evaluation of lazy deletion methods in r-trees. *GeoInformatica*, 7(4):337–354, 2003.

[33] T. Nguyen, Z. He, R. Zhang, and P. Ward. Boosting moving object indexing through velocity partitioning. *PVLDB*, 5(9):860–871, 2012.

[34] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[35] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[36] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: an efficient index for predicted trajectories. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 637–646, 2004.

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

[38] S. Ray, R. Blanco, and A. K. Goel. Supporting location-based services in a main-memory database. In *IEEE 15th International Conference on Mobile Data Management, MDM 2014, Brisbane, Australia, July 14-18, 2014 - Volume 1*, pages 3–12, 2014.

[39] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations, 1994.

[40] A. Sadilek and J. Krumm. Far out: Predicting long-term human mobility. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[41] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. López. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 331–342, 2000.

[42] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[43] D. Sidlauskas, K. A. Ross, C. S. Jensen, and S. Saltenis. Thread-level parallel indexing of update intensive moving-object workloads. In *Advances in Spatial and Temporal Databases - 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24-26, 2011, Proceedings*, pages 186–204, 2011.

[44] D. Sidlauskas, S. Saltenis, C. W. Christiansen, J. M. Johansen, and D. Saulys. Trees or grids?: indexing moving objects in main memory. In *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*, pages 236–245, 2009.

[45] D. Sidlauskas, S. Saltenis, and C. S. Jensen. Parallel main-memory indexing for moving-object query and update workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48, 2012.

[46] D. Sidlauskas, S. Saltenis, and C. S. Jensen. Processing of extreme moving-object update and query workloads in main memory. *VLDB J.*, 23(5):817–841, 2014.

[47] Y. N. Silva, X. Xiong, and W. G. Aref. The rum-tree: supporting frequent updates in r-trees using memos. *VLDB J.*, 18(3):719–738, 2009.

[48] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 611–622, 2004.

[49] 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.

[50] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. Knowl. Data Eng.*, 16(10):1169–1184, 2004.

[51] X. Xiong and W. G. Aref. R-trees with update memos. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 22, 2006.

[52] X. Xu, L. Xiong, V. S. Sunderam, J. Liu, and J. Luo. Speed partitioning for indexing moving objects. In *Advances in Spatial and Temporal Databases - 14th International Symposium, SSTD 2015, Hong Kong, China, August 26-28, 2015. Proceedings*, pages 216–234, 2015.

[53] X. Xu, L. Xiong, V. S. Sunderam, J. Liu, and J. Luo. Vpindexer: Velocity-based partitioning for indexing moving objects. In *SIGSPATIAL 2015 International Conference on Advances in Geographic Information Systems*, 2015.

[54] A. Y. Xue, R. Zhang, Y. Zheng, X. Xie, J. Huang, and Z. Xu. Destination prediction by sub-trajectory synthesis and privacy protection against such prediction. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 254–265, 2013.

[55] G. Yavas, D. Katsaros, Ö. Ulusoy, and Y. Manolopoulos. A data mining approach for location prediction in mobile environments. *Data Knowl. Eng.*, 54(2), 2005.

[56] J. J. Ying, W. Lee, T. Weng, and V. S. Tseng. Semantic trajectory mining for location prediction. In *19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2011, November 1-4, 2011, Chicago, IL, USA, Proceedings*, pages 34–43, 2011.

[57] M. L. Yiu, Y. Tao, and N. Mamoulis. The b^{dual}-tree: indexing moving objects by space filling curves in the dual space. *VLDB J.*, 17(3):379–400, 2008.

[58] M. Zhang, S. Chen, C. S. Jensen, B. C. Ooi, and Z. Zhang. Effectively indexing uncertain moving objects for predictive queries. *PVLDB*, 2(1):1198–1209, 2009.

[59] R. Zhang, H. V. Jagadish, B. T. Dai, and K. Ramamohanarao. Optimized algorithms for predictive range and KNN queries on moving objects. *Inf. Syst.*, 35(8), 2010.

[60] J. Zhou, A. K. H. Tung, W. Wu, and W. S. Ng. A "semi-lazy" approach to probabilistic path prediction. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 748–756, 2013.

[61] Y. Zhu, S. Wang, X. Zhou, and Y. Zhang. Rum+-tree: A new multidimensional index supporting frequent updates. In *Web-Age Information Management - 14th International Conference, WAIM 2013, Beidaihe, China, June 14-16, 2013. Proceedings*, pages 235–240, 2013.