Imperial College London
Department of Computing

Towards ML-enhanced Trajectory Query Processing

Guang Yang

Submitted in part fulfillment of the requirements for the degree of
Doctor of Philosophy in Computing from Imperial College London and
the Diploma of Imperial College, April 2024

Abstract

The rapid growth in location-based services has generated massive trajectory data. Centralized databases have faltered under the demands of low-latency services and stringent privacy regulations. Addressing scalability and flexibility challenges for large-scale trajectory data management necessitates research on optimized index structures.

In this thesis, we propose a modular approach using hybrid indexing to tackle the two challenges from two levels: (1) Global scalability point of view (across machines) and (2) Local efficiency point of view (within each machine). Together, these two levels form a comprehensive view of the modular approach to enable efficient complex trajectory data processing. Specifically, we introduce the concept of machine learning (ML)-enhanced indexes or learned indexes into trajectory indexing. However, directly applying learned indexes introduces challenges since they cannot preserve the continuity of trajectories or cope with the dynamic nature of trajectories (requiring continuous updates).

From our insight into trajectory update patterns, where spatial updates shift the data locations within a predefined region while the temporal dimension continuously grows and expires, we propose to optimize traditional indexing techniques to improve updates in spatial dimensions at the global level. This allows the data structure to be more robust to data distribution shifts. Learned index techniques are applied locally to enhance temporal filtering and continuous timestamp updates.

We present a comprehensive view of our design approach by showcasing solutions to trajectory problems with different requirements. Specifically, at the global level, we developed a distributed approach for efficiently answering distance joins while preserving trajectory continuity and a framework for in-network range queries leveraging sensor and data distribution to ensure privacy. We then optimize each machine by transforming the temporal component of trajectories as a stream indexing problem. Two updatable and parallelizable learned indexes were developed to handle the temporal streams under various query workloads and update characteristics.

Acknowledgements

I would like to express my gratitude to everyone who helped and accompanied me along the way to make the completion of this thesis possible.

First, I like to thank Thomas Heinis, my supervisor, whose expertise, support, understanding, and patience made my Ph.D. journey enjoyable. Your guidance has transformed a civil engineer into a computer scientist. I have learned so much about being a researcher, from the ins and outs of generating ideas and formalizing approaches to applying for funding. Your mentoring is what made this thesis possible.

My sincere appreciation goes to my collaborators Liang Liang, Ali Hadian, Ahbirup Ghosh, Giannis Evagorou, Luis Alberto Croquevielle, and Taiyi Wang. I am truly grateful to get the opportunity to work with you all. I would like to especially thank Liang, with whom I coauthored several papers. Liang and I have worked very closely together since my second year and have since become close friends. Having a collaborator really helped ease my Ph.D. journey. We motivated one another and gained a lot of knowledge from technical discussions to off-topic conversations.

I am also grateful to my examiners, Eiko Yoneki and Peter Pietzuch, whose invaluable insights and feedback made this final thesis possible.

I would also like to thank Amani El-Kholy for taking care of all the administrative problems I had to face and all those who I've worked with throughout my teaching assistant career: Josiah Wang, Tom Crossland, and Mark Van Lokeren.

I am also thankful for my friends, both within and outside of the research community, who have been by my side on both good and bad days. Apologies for not explicitly listing your names.

Finally, and above all, I am deeply grateful to my family, who has provided me with endless love and support throughout my life. To my parents, Hai Li and Peigang, for being the best parents I could ever wish for. Also, to my aunt, Peiying, for the unconditioned support and guidance throughout my life.

This thesis is based on the collective effort and kindness of all who have been a part of my life during this period. Thank you very much.

Dedication

This thesis is dedicated to my parents and my aunt. I could not have accomplished any of this work without their love, support, and guidance throughout my life. Thank you for all your sacrifices and support, which paved the way for my achievements; this thesis is dedicated to you!

Statement of Originality

This thesis includes my original work completed during my Ph.D. studies at the Department of Computing at Imperial College London. All joint authorships are clearly stated and acknowledged in Section 1.5, and their contributions are specified. I declare that the research presented in this thesis represents my original work.

Copyright Declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-NonCommercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Contents

Abstract **i**

Acknowledgements **iii**

Statement of Originality **vii**

Copyright Declaration **ix**

1 Introduction **1**

1.1 Motivation and Objectives 1

1.2 Challenges with Existing Approaches 3

1.3 Contributions 4

1.4 Thesis Structure 6

1.5 Publications 7

2 Background **8**

2.1 Overview 8

2.2 Query Processing 8

2.2.1 Background Concepts in Query Processing 8
2.2.2 Spatial Query Processing 9
2.2.3 Temporal Query Processing 11
2.2.4 Spatiotemporal Query Processing 12
2.2.5 Trajectory Query Processing 13
2.3 Indexing 15
2.3.1 Background Concepts in Indexing 15
2.3.2 Indexing Trajectories 16
2.3.3 One-dimensional Indexes 17
2.3.4 Spatial Indexes 21
2.3.5 Multidimensional & Trajectory Indexes 24
2.4 Learned Indexes 28
2.4.1 Introduction to AI & DB 28
2.4.2 Introduction to Learned Indexes 30
2.4.3 Design of Learned Indexes 32
2.4.4 Baseline Learned Indexes 36

3 Distributed Trajectory Distance Join 38

3.1 Overview 38
3.2 Drawbacks of Existing Works 39
3.2.1 Trajectory Joins Algorithms 40
3.2.2 MapReduce-based Trajectory Joins 41

3.3 Background 42
3.3.1 Trajectory Data Model 42
3.3.2 Query Formulation 43
3.4 Trajectory Distance Join Algorithm 46
3.4.1 Design Overview 46
3.4.2 The Partitioning Phase 47
3.4.3 The Filtering Phase 49
3.4.4 The Refining Phase 52
3.4.5 Time Window Distance Join and Exact Distance Join 54
3.4.6 Time Match Exact Distance Join 55
3.5 Experimental Evaluation 55
3.5.1 Experimental Setup 55
3.5.2 System Parameters Evaluations 56
3.5.3 Query and Data Parameters Evaluation 59
3.5.4 Scalability Evaluation 60
3.6 Summary 62

4 In-Network Privacy-aware Range Queries 63

4.1 Overview 63
4.2 Drawbacks of Existing Works 66
4.2.1 Centralized Approaches 66
4.2.2 In-Network Approaches 67

4.2.3 Sensor Selection 67

4.3 Background 67

4.3.1 Problem Setting 68

4.3.2 Design Reasoning 69

4.3.3 Mobility & Sensing Domain 70

4.3.4 Planar Graph Representation 71

4.3.5 Query Formulation 72

4.3.6 Discrete Differential Forms 72

4.4 Privacy-aware Range Query Framework 74

4.4.1 Overview 74

4.4.2 Constructing Planar Graphs 75

4.4.3 Query Oblivious Sensor Selection 75

4.4.4 Query Adaptive Sensor Selection 77

4.4.5 Generating the Sampled Graph 81

4.4.6 Dispatching Queries 82

4.4.7 Updating the Sampled Graph 83

4.4.8 Querying the Sampled Graph 85

4.4.9 Computing the count function using regression models 86

4.5 Experimental Evaluation 87

4.5.1 Experimental setup 88

4.5.2 Size of the Sampled Graph 90

4.5.3 Query Region Size . . . 90
4.5.4 Communication Cost & Speed up . . . 91
4.5.5 Queries missed . . . 92
4.5.6 Upper bound approximation . . . 92
4.5.7 kNN based connectivity . . . 93
4.5.8 Performance of regression model . . . 94
4.6 Summary . . . 94

5 An Updatable Learned Index for Temporal Streams 95

5.1 Overview . . . 95
5.2 Drawbacks of Existing Works . . . 97
5.3 A Fast Learned Index for Rolling Time-frames . . . 98
5.3.1 Overview . . . 98
5.3.2 Index Structure . . . 100
5.3.3 Search . . . 101
5.3.4 Enqueue . . . 102
5.3.5 Dequeue . . . 103
5.3.6 Auto-tune . . . 104
5.4 Parallelizing FLIRT . . . 105
5.4.1 Parallel Partitioned Flirt (PPFlirt) . . . 106
5.4.2 Parallel Shared Flirt (PSFlirt) . . . 109
5.5 Experimental Evaluation . . . 110

5.5.1 Experimental Setup . . . 110
5.5.2 Evaluation of the auto-tuner . . . 112
5.5.3 Comparison with the baselines . . . 113
5.5.4 CPU-level Micro-Analysis . . . 115
5.5.5 Window Size . . . 116
5.5.6 Number of Iterations . . . 117
5.5.7 Range Query . . . 117
5.5.8 Read/Write Ratio . . . 118
5.5.9 Parallel FLIRT Performance . . . 118
5.5.10 Query Dispatcher Throughput . . . 120
5.6 Summary . . . 121

6 A Memory-efficient Learned Index for Key-based Streams 122

6.1 Overview . . . 122
6.2 Index-Based Window Processing (IBWP) . . . 124
6.3 Drawbacks of Existing Work . . . 124
6.4 The Sliding Window Learned IndeX . . . 125
6.4.1 Overview . . . 125
6.4.2 Index Structure . . . 127
6.4.3 Search . . . 129
6.4.4 Insert . . . 130
6.4.5 Delete . . . 131

6.4.6 Bulk-load . . . 132
6.4.7 Retrain . . . 133
6.4.8 Analytical Cost Model . . . 135
6.4.9 Auto-tuner . . . 136
6.5 Parallel SWIX . . . 137
6.5.1 Index Structure . . . 137
6.5.2 Parallel Search . . . 138
6.5.3 Parallel Insert & Delete . . . 139
6.5.4 Concurrent Retrain . . . 139
6.6 Experimental Evaluation . . . 140
6.6.1 Experimental Setup . . . 141
6.6.2 Self Analysis . . . 143
6.6.3 Summary of Comparative Analysis . . . 144
6.6.4 Window Length . . . 146
6.6.5 Scan Range . . . 147
6.6.6 Skewness in Timestamp . . . 148
6.6.7 Read/Write Ratio . . . 150
6.6.8 Ordered Workload . . . 150
6.6.9 Parallel Learned Indexes . . . 151
6.7 Summary . . . 153
6.8 Limitations of FLIRT and SWIX . . . 154

6.8.1 Limitations of Flat Index Structures 154

6.8.2 Limitations in Stream Processing 155

7 Conclusion **157**

7.1 Summary of Thesis Achievements 157

7.2 Future Work 159

Bibliography **164**

List of Tables

3.1 Optimal configuration for each distance join method. 59

5.1 Notation table for Chapter 5 100

List of Figures

2.1 The database distribution of a sorted array of keys is represented as a cumulative distribution function (CDF). 31

3.1 *Right:* False positives caused by the query radius not intersecting with endpoints. *Left:* Example when joining two trajectories P and Q. 41

3.2 Example of minimum time matched distance (red). Segments in Q (black) are matched with segment $\overline{p_i p_{i+1}}$ (blue) between $[t_i, t_{i+1}] \in P$. Segments or parts of the segments that are outside of the duration are not considered (grey). *Left* figure shows the distances with the time dimension, and the *Right* figure shows the results in the spatial dimensions only. 45

3.3 Example showing the difference in the number of grid intersections between segments and the line segment. A segment can spatially intersect 7 grid cells and 3 time slices, while a line segment intersects at most 3 grid cells and 2 time slices. 46

3.4 Performance of distance join with varying grid resolution. 56

3.5 Run time breakdown of distance join with varying grid resolution. 56

3.6 Performance of distance joins with varying number of time slices 57

3.7 Run time breakdown. 57

3.8 Query and Data Parameters Evaluations 59

3.9 Performance of distance join with increasing dataset size. 61

3.10 Performance with increasing number of nodes. (a) Total Time. (b) Average Execution Time per Node. 61

4.1 (a) A sensors network tracking two users u_b and u_r. (b) The associated communication infrastructure. 64

4.2 Example of road network: the mobility graph is $\star G$, dual graph G is the sensing graph. Two objects, u_b and u_r, are tracked within the region. Q_R is the query region (green). u_b enters the Q_R at 10:00 and leaves at 10:15. u_r enters the Q_R at 10:05 and does not leave. 71

4.3 (a) 0,1,2 cells (b) An oriented face σ 72

4.4 An overview of the framework. 73

4.5 Sampling communication sensor location using road network of Beijing. (a) uniform sampling. (b) systematic sampling. (c) stratified sampling. (d) *kd*-tree sampling. (e) quadtree sampling. (f) Regions (red) selected by submodular maximization given a set of queries. 76

4.6 Example of two overlapping query regions. (a) Q_1 and Q_2 are two overlapping query regions to form $Q_3 = Q_1 \cap Q_2$. (b) Disjoint query regions $Q_1 - Q_3$, $Q_2 - Q_3$ and Q_3. 79

4.7 Generating edges from sampled nodes using triangulation and kNN (k=3). (a) Edges after triangulating $\tilde{V}$. (d) Edges after finding k-nearest neighbor for each node v in $\tilde{V}$. (b,e) Replacing edges within the shortest path in G. The blue nodes are intersection nodes added to ensure $\tilde{G}$ is planar. (c,f) $\tilde{G}$ after simplification. . 81

4.8 The query region Q_R (green) defines the rectangular query region as faces in the sensing graph G. R_1 and R_2 are the upper and lower approximation of Q_R. . . . 82

4.9 Differential form example. (a) The road network is the dual graph (dotted line), with the sensor network as the primal graph (solid line). $\star v_{ext}$ is the infinity node that acts as a source and sink for objects entering and leaving the mobility graph. (b) Target T moving from face σ to τ. The query region Q_R is the union of faces σ and τ ($Q_R = \sigma \cup \tau$). (c) An alternative example of three trajectories moving in and out of σ at different timestamps. 83

4.10 Storing the range count and timestamp(s) in an example tracking form using popular regressors. 86

4.11 Lower bound error of static queries. *(a)* w.r.t graph size *(b)* w.r.t query sizes. . . 89

4.12 Lower bound error of transient queries *(a)* w.r.t graph size. *(b)* w.r.t query sizes. 89

4.13 *(a)* The number of nodes accessed w.r.t query sizes. *(b)* Execution time. 91

4.14 *(a,b)* The number of queries missed over the total number of queries. *(c,d)* The upper bound relative error. 92

4.15 *(a)* The lower bound relative error of using kNN-based connectivity. *(b)* The number of edges accessed in the sampled graph w.r.t to G. 93

4.16 *(a,b)* The additional error generated by the regression models. *(c)* The storage cost of edges. 93

5.1 search vs update latency spectrum for sliding window processing. 96

5.2 Overview of a monotonic key stream workload indexed by FLIRT. 98

5.3 Structure of FLIRT illustrating the workings of a search operation. 100

5.4 Example of an enqueue. (a) Segmentation algorithm (b) Insertion by appending. (c) Insertion by splitting. 102

5.5 Example of a dequeue. (a) Deleting a record in a non-empty segment. (b) Deleting an empty segment. 103

5.6 Structure of Parallel Partitioned Flirt (PPFlirt), where a thread manages its partition. The enqueue thread (initially empty) is colored in blue. 106

5.7 Enqueue and Dequeue in PPFlirt. 106

5.8 Asynchronous search in PPFlirt. 107

5.9 Structure of Parallel Shared Flirt (PSFlirt), where all threads share a common "global" index. The update thread (in blue) only accesses the first and last segments. 108

5.10 Comparison between auto-tuned and static error thresholds. 112

5.11 Comparative analysis of indexes against FLIRT. 113

5.12 Processor level micro-analysis. 115

5.13 Comparative analysis for increasing window sizes. 116

5.14 Comparative analysis for simulating continuous updates. 116

5.15 Comparative analysis using range queries. 117

5.16 Comparative analysis under varied read-to-write ratios. 118

5.17 Comparison of concurrent variants of FLIRT with single-thread FLIRT (in green) for uniform query workloads. 119

5.18 Comparison of concurrent variants of FLIRT for skewed query workloads. 119

5.19 Comparative analysis of query dispatchers. 120

6.1 A sliding window for index-based window processing. 123

6.2 Drawbacks of existing indexing in an IBWP workload. 124

6.3 Index structure of SWIX 126

6.4 Illustration of search, insert, and delete operations using SWIX. The search process is shown in green, where we employ two *predict-correct* steps to find the key 700. We search the buffer since the key is not in the array. The expired data (key 630) is implicitly deleted in this process, shown in red. We then insert the same key at position 4, which causes a shift (shown in blue). 128

6.5 Different cases of segment modification during insertion. 129

6.6 Example of second derivative segmentation algorithm for cold starts. 132

6.7 The steps for a rendezvous batch retrain of segments. 133

6.8 (a) Structure of Parallel SWIX. (b) Synchronization during meta update. (c) Synchronization during meta retrains. 137

6.9 Comparison between auto-tuned and static *tol*. 143

6.10 ϵ_{seg} evolution over updates. 144

6.11 Memory overhead $\frac{total\ size - data\ size}{data\ size}$ and latency $\frac{total\ time}{total\ updates}$ for each experiment setting excluding the ordered workload. For memory overhead, MAX is measured when the total memory consumption (*total size*) is at its largest and can be smaller than the AVG. 145

6.12 Analysis of different window sizes and scan ranges. 146

6.13 Latency (μs) and Index footprint (MB) for different degrees of σ^2 in timestamps. 148

6.14 Comparison of learned indexes on different workloads. 149

6.15 Comparison of parallel learned indexes with random insertion. The first three figures (from the left) indicate the throughput for different read-write ratios. The fourth figure shows the memory usage. 151

6.16 Comparison of parallel learned indexes with ordered insertion. 152

Chapter 1

Introduction

1.1 Motivation and Objectives

Advancements in navigation, wireless communication, and data collection systems have produced a massive volume of data that are increasingly crucial in large-scale online location-based services [182, 185]. These location-based data describe the movements of objects (e.g., humans, vehicles, animals, and natural phenomena [208]). They are represented with a series of spatial locations at different time instances, known as trajectories. NASA produces over 4TB of stars and asteroid trajectories a day [11], the New York City Taxi and Limousine Commission have produced over 1.1 billion taxi trajectories [48] and the Danish Maritime Authority has collected over 1.8TB of vessel trajectories [20]. These trajectory data have been adopted in many scientific research such as traffic flow analysis [194], human mobility pattern studies [87], and environmental monitoring [196]. They are also widely used commercially in location-based services. For example, Ride-sharing companies analyze user behavior from trajectories to improve travel time predictions, demand forecasting, and carpool scheduling protocols [58].

As the volume of datasets and the complexity of queries continue to grow, modern databases face two significant challenges in terms of scalability and flexibility. When it comes to scalability, low-latency services (e.g., Uber ride find and Google Maps route search) demand real-time query processing whilst processing multiple queries concurrently. Traditional centralized

database solutions are no longer sufficient for these requirements [8]. This calls for more scalable frameworks that can handle more data at a higher processing rate.

On the other hand, flexibility is key for analytics in modern applications where systems have to process complex queries with constraints. For example, trajectory data is considered sensitive data since it records users' daily movement patterns. As user awareness of data privacy and data protection legislation (e.g., GDPR) have increased over the years, database systems must evolve to ensure data privacy and security while storing and processing these datasets [70].

The most challenging part of the problem is that modern databases cannot deal with these two challenges simultaneously. Bridging these demands requires scalability and flexibility to be considered when designing data structures at the storage level. For example, distributing data across nodes allows for scalable processing, but the system cannot deal with queries requiring trajectory continuity or adapt quickly to the space-time heterogeneity in data distribution. This calls for redesign of the underlying storage formats and index structures to support heterogeneous data models and complex query operators. While redesigning, we must consider the trade-off between the two demands, where real-time analytics require low latency data structures that efficiently manage ever-growing high-velocity data. At the same time, data privacy compliance measures require secure access methods (e.g., encryption or fine-grained access control), which hinders response time. Therefore, there are no one-size-fits-all solutions, which calls for a more layered, modular approach, where each layer can be tailored to meet specific requirements.

More formally, we hypothesize that *"To support large-scale trajectory data management, a multi-layered, modular approach to designing trajectory storage formats and index structures will scale better (scalability) across different query and system requirements (flexibility) compared to one-size-fits-all data structures."*.

1.2 Challenges with Existing Approaches

Recently, the idea of instance-optimized design philosophy [55, 54] has gained much attention by allowing database components to be infinitely configurable (using parameters) to any workload. This differs from traditional design philosophy, which seeks to minimize the complexity presented to the user. The traditional viewpoint is to achieve good performance for the average workload and reduce the chance of finding undesirable configurations. An instance-optimized design instead seeks to open up the configuration space to allow the database component to adapt to specific workloads and avoid undesirable configurations by using algorithms (e.g., ML) to adjust the parameters automatically.

For indexing structures, the instance-optimized design has been applied to learned indexes [111], which were proposed to provide a more compact and efficient query structure than the conventional algorithmic indexes. The idea is to use machine learning models to "automatically" adjust the index parameters according to the data distribution. Specifically, the model predicts the address in memory (or physical storage) from a key before searching around the predicted location to find the exact address. The efficiency comes from two factors: first, the model prediction is faster than the search algorithm (e.g., binary search). Second, the retrieval speed is improved if the model can give an accurate prediction close to the actual address.

Although learned indexes have shown to be effective for certain workloads and datasets, especially read-only one-dimensional datasets [111, 109, 72], their application to trajectories is still a challenge. First, trajectory data contain spatial, temporal, and semantic information, which requires data to be organized sequentially and often grouped [15]. Multidimensional learned indexes have been proposed to handle point datasets and queries [139, 57, 49]. A trajectory under these indexes is scattered across the data structure, where points are grouped based on spatial proximity. This ignores the continuity of trajectories, and we lose information about movement between points [145]. Learned indexes on non-point data [190] only support spatial data and topology queries, making them ill-suited for indexing trajectories. Second, trajectories are constantly updated with new points that need to be related to previous points from the same trajectory, which is not fulfilled by multidimensional learned indexes only supporting

spatial point updates [120, 137]. Some indexes use space-filling curves to transform the problem into one dimension [148, 79]. This is inefficient since the constant temporal updates in trajectories require the space-filling curve to be rebuilt for all dimensions, even if the spatial locations do not change. Furthermore, updates to learned indexes generate inaccuracies in the model, which calls for retrains to ensure performance. Therefore, we can conclude that, at this current state, it is challenging to compel the case of learned indexes as a replacement to traditional trajectory indexes, nor is it a one-size-fits-all solution to tackle both scalability and flexibility.

1.3 Contributions

This thesis proposes a modular approach by exploring hybrid indexing that combines learned and traditional indexing techniques to tackle scalability and flexibility challenges. We do so by separating the problem into two levels of optimization. First, from a global scalability point of view, a trajectory index must efficiently store and process a high volume of data across multiple machines. The second level is from a local efficiency perspective, where indexing techniques are used to optimize (speed up and save space) the high velocity of data in each machine.

At a global level, we focus on ensuring that the index structure can scale well with data. The characteristics and update frequency of trajectories dictate that it is challenging to design a model that learns spatial proximity and preserves continuity at a global level. Specifically, the spatial domain is usually confined within fixed regions (e.g., within the city's boundary), whilst the temporal range continuously evolves with new and expiring timestamps, which results in the distribution of spatial objects shifting around in a fixed area over time. This makes it difficult for a model to infer the position of objects, as the model "snapshots" the distribution during training. Constant retraining must be done, which hinders our scalability and latency requirements. Therefore, we use traditional indexing techniques to accommodate the high volume and speed while remaining robust to distribution changes across multiple machines.

At a local level, we optimize for efficient updates and queries in the rapidly evolving dimension.

For trajectories, this is the temporal dimension, which exhibits characteristics of streaming workloads. Learned indexes excel for this domain as they can exploit the data distribution's predictability to reduce the indexes' latency and storage size.

In this thesis, we aim to answer the following questions:

- At a global level, how can we accommodate the characteristic of trajectory updates to efficiently handle queries across multiple machines using traditional indexing techniques?
- At a local level, how do we adapt learned indexes to handle the update-intensive temporal component of trajectories and exploit the predictability in the temporal dimension?

To provide a holistic solution to handling complex trajectory queries in large-scale data management, this thesis answers the two questions from different perspectives in terms of the system environment and type of queries. Each chapter introduces a module that tackles a specific requirement and provides a perspective on the overall solution.

Specifically, to answer the first question, we focus on two problems: distributed distance join (Chapter 3) and in-network privacy-aware range queries (Chapter 4). The distance join problem focuses on reducing the impact of data partitioning updates across multiple machines while ensuring trajectory continuity. The in-network privacy-aware range query showcases how to efficiently handle queries when data is scattered in a network of sensors. The privacy-aware requirement makes this problem more challenging as we need to ensure no third parties have access to the full trajectories at any time.

Chapter 5 and 6 tackles the second question by framing the updates in the temporal dimension as a streaming indexing problem. Chapter 5 introduces FLIRT, a learned index optimized for update-heavy streaming workloads where the keys are monotonically ordered, which can be directly used in the in-network framework introduced in Chapter 4. We then introduced SWIX in Chapter 6 for more generalized streaming queries where the updates can occur randomly (e.g., filter by time and match trajectories based on primary keys, in this case, line segment IDs). Both FLIRT and SWIX support parallel processing, making them effective for handling long temporal ranges in trajectory query processing.

Together, these solutions highlight the importance of integrating scalability and flexibility when designing the underlying storage format and index structure. We aim to demonstrate how our modular approach can substantially improve large-scale trajectory data management by providing a comprehensive view across system environments and query types.

1.4 Thesis Structure

The structure of the thesis is as follows:

Chapter 2 introduces all relevant concepts and background information on trajectory query processing, traditional, and learned indexes.

Chapter 3 presents the framework for efficiently answering trajectory distance joins. We introduce three variants of the trajectory distance join. The framework is based on a distributed approach on Spark. We show how storing trajectories as line segments preserves the continuity of trajectories and allows information between points to be interpolated. Using a distance-correlated grid-based partition and indexing, the framework is experimentally shown to outperform the state-of-the-art point-based approach.

Chapter 4 introduces the framework for efficient privacy-aware in-network aggregate queries. The data is stored and queried within the sensor network to ensure no third parties have access to the full trajectories at any time. The framework uses a modular design where the spatial domain is decomposed using a planar graph representation of the in-network system. The time domain stores the counts of objects as a sequence of timestamps. We show the feasibility of using constant-sized machine-learning models to index the time sequence.

Chapter 5 presents FLIRT, a parameter-free index structure for managing a sliding window over a high-velocity stream of data where keys are monotonically ordered. We show how FLIRT is optimal for indexing time-series data with a high rate of insertion and deletion commonly seen in the temporal dimension of trajectory workloads.

Chapter 6 presents SWIX, a memory-efficient sliding window learned index that supports

random insertions. Unlike FLIRT, SWIX does not require the keys to be monotonically ordered and allows keys to expire with time. This makes it ideal for workloads where data arrives and expires based on timestamps but is searched using a primary key.

Chapter 7 concludes the thesis and presents future works.

1.5 Publications

The list of publications that are included in this thesis are (**bold** represent first author(s)):

1. **Guang Yang** and Thomas Heinis. Distributed Spatiotemporal Distance Join for Trajectory Data. In *2021 IEEE 17th International Conference on eScience (eScience)*, 2021. This work was also featured in a talk with the Royal United Services Institute (RUSI) and the Imperial College London's Institute for Security Science and Technology [43].

2. **Guang Yang**, Ahirup Ghosh, Liang Liang, and Thomas Heinis. In-Network Approximate and Efficient Spatiotemporal Range Queries on Moving Objects. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2024.

3. **Guang Yang**, **Liang Liang**, Ali Hadian, and Thomas Heinis. FLIRT: A fast learned index for rolling time frames. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2023.
   Liang Liang and Guang Yang contributed equally to this paper. Both contributed equally to the design of FLIRT. I implemented the code and experiment for FLIRT.

4. **Liang Liang**, **Guang Yang**, Ali Hadian, Luis Alberto Croquevielle, and Thomas Heinis. SWIX: A Memory-efficient Sliding Window Learned Index. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2024.
   Liang Liang and Guang Yang contributed equally to this paper. Specifically, we both contributed equally to the conceptual, and theoretical design and development of SWIX. I led the implementation of SWIX and conducted the experimentation.

Chapter 2

Background

2.1 Overview

In this chapter, we present the necessary background information for this thesis. We first introduce the relevant concepts in query processing, focusing on spatiotemporal and trajectory query processing. We then examine how indexing structures improve query processing and present common trajectory indexes. Lastly, we briefly discuss how AI is integrated into databases before focusing on the concept of learned indexing, where machine-learning techniques are integrated into indexing structures to improve performance and save space.

2.2 Query Processing

2.2.1 Background Concepts in Query Processing

We begin by explaining the basic concepts of query processing. Querying aims to retrieve or update information from a storage medium using a database management system (DBMS). The process involves three phases [170] in the following order:

1. *Parsing and translation:* The first step is to convert the query into an executable represen-

tation. Queries are usually written in high-level query languages such as SQL. Therefore, a parser must convert the inputted strings into tokens (e.g., keywords and operators). The parser also checks the validity and syntax of the query. E.g., the parser validates the query against the data schema to ensure the tables and columns exist.

2. *Optimization:* In the second stage, the DBMS determines the best query execution plan to minimize the required resources (e.g., CPU clocks, memory usage, or I/O). It may use techniques such as heuristic rules, statistics about the data, and logical transformations to perform the optimization [154].

3. *Evaluation:* Once the best execution plan has been generated, the DBMS performs the operations (e.g., search, sort, and/or update) and returns the result. The challenge here is the complexity of query evaluation due to the diverse data types, data sizes, and query requirements. This led to the development of algorithms tailored for different query operations. For example, binary search is used for the `select` operation on one-dimensional sorted data [147], hash joins are developed for efficient join operations on unsorted data [90], and Quicksort is designed for sorting on non-linked list data [99].

We now focus on spatiotemporal operations and review the related works. Spatiotemporal query processing handles and analyzes spatial (location-based) and temporal (time-based) data. We explain the spatial and temporal components separately before introducing spatiotemporal query operations.

2.2.2 Spatial Query Processing

Spatial data defines the static objects in space. The objects can take the form of points, lines, and polygons. For example, a spatial point with a longitude and latitude can be used to represent a city. These are stored in the form of raster representations or vectors. The essence of the spatial query class is to capture the spatial relationship between the objects. These relationships are encompassed by spatial operations divided into geometric and Boolean-based categories [15].

Geometric operations retrieve information about spatial objects using their spatial geometry and dimension. For example, queries such as the shortest distance between points, the length of a line, or the area of a polygon all return information based on geometry. Geometric operations can be used to manipulate spatial objects to create new geometries. For example, the minimum bounding rectangle and union operations can used to create objects from polygons.

Boolean operations capture more complex relationships between spatial objects by using a spatial predicate to encompass their relationships. These operations take the predicate as an argument to filter out objects. Therefore, the results are a subset of objects from the database. The predicates are topology, directional, and distance-based [76].

- *Topology-based* predicates capture the topology-relations such as equals, intersects, and range selection.

- *Directional-based* predicates find objects based on directional location from a reference point or object. For example, find points *left* of a region.

- *Distance-based* operations find objects based on proximity (e.g., distance search and closest pair of objects). The proximity is defined using spatial distances such as Euclidean or Manhattan distances. Two widely studied spatial operations, the k-nearest neighbors [52] and spatial joins [104], are considered distance-based Boolean operations.

More complex operations can be built on top of geometric and Boolean operations. For example, class-based operations such as point-to-polygon queries [159] capture the interactions between two object classes. Another example is to combine the nearest neighbor and join operators into a k-NN join [210]. Other data sources can also be combined with spatial data to generate more complex queries. One of the most well-known examples is the spatial-textural queries [40] combining textual and spatial predicates. For example, the spatial keyword query takes a set of locations and keywords to find all locations that contain the set of keywords [36].

2.2.3 Temporal Query Processing

Temporal data represents a time point or interval for a specific object [160] (or attribute of the object). For example, the date and time when the stock price is recorded. Each data point is stored as a timestamp. Semantically, temporal data is always ordered by time and forms an order sequence known as a time sequence. Depending on the application, the time sequence can have step-wise constant, discrete, or continuous values [143]. Briefly speaking, step-wise constant values are used to describe objects with sudden changes. Discrete values are used to represent timestamps collected periodically. Continuous values are used to model constantly changing objects.

The difference between temporal data and key-based one-dimensional sequential data is that temporal data processing tries to understand temporal relationships, such as considering the evolution of the data and evaluating trends over time. Time-based DBMS uses concepts such as bi-temporal data models where two-time axes (valid and transaction time) allow objects to be queried at different states in times [105]. Non-temporal sequential data processing cannot answer such queries as values are overwritten when updated.

Temporal operations are based on comparing time intervals, each comprising a start and end time. The operation is based on thirteen basic temporal relations, details in [14]. Temporal query processing takes a similar approach to spatial Boolean queries, where we filter a subset of the data based on a given temporal operation. We can break temporal queries into general, aggregation, and joins.

- *General temporal queries* encompass temporal relationships using temporal operations and return a subset of the data. For example, a range query would find all objects between a time interval. A select query may want to find events with the same start time and last for less than d duration. One of the most widespread queries is the snapshot or time travel query [106], which returns a view of a past data version.
- *Temporal aggregation* groups data based on running time points or intervals. The result is an aggregate value. E.g., calculate each month's average daily price of stock X.

- *Temporal Joins* add temporal relations to a non-temporal join. It takes two predicates, the original join predicate based on the key and one of the temporal operations [100]. For example, finding the companies with stock prices higher than x between time intervals t_1 and t_2 that also appeared on the news.

Similar to spatial query processing, the complexity of the query depends on the data and query requirements. Stream processing stems from temporal query processing to process continuous data streams. The data streams add complexity as data is actively generated and discarded, requiring continuous updates. These queries typically also require real-time query processing. Stream processing typically uses window-based processing [86], where queries are performed over a window of time. For example, find the average stock price in the last hour for every hour starting now. The time window is one hour. We will discuss this in detail in Chapter 5.

2.2.4 Spatiotemporal Query Processing

Spatiotemporal data can be viewed as a temporal evolution of spatial objects [158]. The key feature is that spatiotemporal objects record how spatial objects change continuously with time. These changes include movement, shape transformations (e.g., expanding or shrinking), splitting, merging, appearing, and disappearing [67]. There are many approaches to expressing spatiotemporal objects. For example, the spatio-bitemporal approach [63] stores bitemporal elements alongside the spatial elements. On the other hand, spatiotemporal objects can be stored as special instances of a temporal model [66]. Regardless of the approach, a spatiotemporal object can be geometrically viewed as a spatial object with an additional time dimension. For example, a three-dimensional spatiotemporal object consists of a two-dimensional spatial object with the time axis as the third dimension.

A spatiotemporal query involves spatial and temporal operations in the same framework. Typically, spatiotemporal queries are built upon spatial queries with an additional temporal predicate [10]. We provide a list of fundamental queries that represent the basic query elements in a spatiotemporal system:

- *Object-based queries* answer queries on specific objects with discrete or continuous time sequences. For example, we can combine geometric predicates to find the area of the storm at time t.

- *Spatiotemporal nearest neighbor queries* find all objects spatially and temporally close to the reference object. Spatial closeness is defined by distance, and temporal closeness can be based on a timestamp or time interval. E.g., find the five closest taxis to King's Cross Station now or find the three nearest coffee shops that opened in the last hour.

- *Range queries* retrieve a subset of objects within a specific area or time interval. For example, find all vehicles in Kensington and Chelsea between time t_1 and t_2.

- *Spatiotemporal joins* take two or more datasets and find all objects that meet a spatial and temporal predicate. For example, we join the emergency vehicle dispatch data with accident data to find all emergency vehicles that are dispatched to the accident scene that are less than 1km away within a 10-minute window.

- *Spatiotemporal behavior queries* [143] analyze an object with a unary operator, which aggregates the results in the temporal dimension. For example, find the average speed of a taxi during rush hour.

- *Spatiotemporal aggregate queries* perform aggregation operations (e.g., count, sum, average, min, and max) on spatiotemporal objects. For example, find the average number of vehicles passing through a particular intersection every hour in the last week. It can be combined with range querying to form aggregate range queries. For example, find the total number of vehicles in an area for every hour last week.

2.2.5 Trajectory Query Processing

We now introduce a special type of spatiotemporal data that specifically focuses on capturing the movement history of moving objects known as trajectories. The biggest difference between spatiotemporal data and trajectory data is that the movement is assumed to be continuous.

Although the motion is continuous, trajectories are characterized and stored as sequences of spatiotemporal points due to how the data is collected. Formally, each spatiotemporal point p_i takes the form of (x_i, y_i, t_i). x_i and y_i are spatial positions along the x and y axis, where the axis can be any spatial axis, such as latitude and longitude. t_i represents the timestamp of that position. The point p_i can be extended to higher dimensions. For example, p_i takes the form (x_i, y_i, z_i, t_i) for four dimension. A trajectory T is defined as a sequence of time-ordered spatiotemporal points $\{p_1, \ldots, p_i, \ldots, p_n\}$ such that $\{t_1 ... < t_i ... < t_n\}$.

A trajectory can also be broken down and stored as segments or sub-trajectories. Formally, a segment is defined as the portion of a trajectory such that the timestamps of the segment are within $[t_1, t_n]$. A segment between p_i and p_j is denoted as $s_{ij} = [p_i, p_j]$, where $t_i \geq t_1$, $t_j \leq t_n$ and $t_i < t_j$. A special type of segment is known as a line segment which spans between two consecutive points, mathematically defined as $l_{i,i+1} = \overline{p_i p_{i+1}} = [p_i, p_{i+1}]$. The overline notation shows that the line segment is a vector between two points rather than connecting two discrete points. Therefore, the motion in a line segment can be found using interpolation. Linear interpolation is the most common type of interpolation [68], where we assume that objects move at a constant speed between two consecutive points.

Since the movement of trajectories is continuous, the distance between trajectories needs to account for the temporal closeness. For example, two points may be spatially close but temporally distant. Elements such as time shifting and non-uniform sampling rates between trajectories make comparing trajectories even more challenging. Therefore, when comparing trajectories, an additional level of complexity is required on top of standard distance measures (e.g., Euclidean distance). There is a large volume of research on new distance measures to account for different aspects of trajectory comparisons, such as shape, rotation, and temporal information. Briefly speaking, the distance measures can be broken down into point-to-trajectory and trajectory-to-trajectory comparisons [193]. Point-to-trajectory comparisons are used when the queries are based on points. Trajectory-to-trajectory comparisons match trajectories by considering the shape similarities of trajectories using distance measures such as the dynamic time warping, edit, and Fréchet distances. These are commonly known as similarity measures. Exploring distance measures is not the focus of our work; we refer the readers to [172, 193].

Any spatiotemporal query introduced in Section 2.2.4 can be transformed into trajectory queries by replacing spatiotemporal objects with trajectories and predicates with trajectory-specific distance measures mentioned above. For example, the nearest neighbor query finds all trajectories given a query trajectory using a Fréchet distances for closeness. In addition, there are also trajectory specific queries, such as the shortest path query, which finds the trajectories that connect two query locations with the shortest distance. Our focus in this thesis is the range query and distance join, and we shall define them here:

Trajectory range query: Given a trajectory dataset T, a spatial region R and a time interval $[t_0, t_1]$, find all trajectories segments $s_{ij} \in T$ that intersect with R within the time interval $[t_0, t_1]$. Specifically, we are interested in the trajectory range count query, which aggregates the counts to improve privacy. Formally, a trajectory range count query finds counts all trajectories segments $s_{ij} \in T$ that intersect with R within the time interval $[t_0, t_1]$, where we need to ensure each trajectory is counted once. We formalize this problem in Chapter 4.

Trajectory distance join: Given two trajectory datasets P and Q, and a predicate $pred()$, finds all pairs of trajectories such that there is at least a pair of points (p_i, where $\forall p_i \in P$ and q_j, where $\forall q_j \in Q$) that satisfy the predicate $pred(p_i, q_j) = \text{true}$. The predicate usually involves only the spatial dimension. For example, $dist(p_i, q_j) \leq \epsilon$ where $dist()$ is any spatial distance function and ϵ is the distance threshold. In Chapter 3, we shall discuss the challenges of incorporating temporal components into the predicate.

2.3 Indexing

2.3.1 Background Concepts in Indexing

In this section, we introduce how to process the queries above efficiently. We first introduce the background concepts of indexing before exploring existing index structures for trajectories. The purpose of database indexes is to improve the retrieval speed during a query. In essence, the fate of all queries is to be reduced to retrieving x number of records from a storage medium.

The idea of index structures is to avoid scanning each record in the data during retrieval by mapping a key (the search criteria) to the location of data in the storage medium. The index structure can also contain additional information, such as other attributes in the row, known as a payload.

Index structures are broken down into clustered and non-clustered indexes. For a clustered index, the physical location of the data has the same order as the index. Therefore, there can be only one clustered index for each database table. Clustered indexes are also known as primary indexes, and their keys are referred to as primary keys. For non-clustered indexes, the ordering of the data and the index does not have to be the same. The index would instead save a pointer to the location alongside the key. Therefore, multiple non-clustered indexes can be built for each table. These are also referred to as secondary indexes.

A vast number of index structures have been developed for different types of queries. The choice of the index depends not only on the query but also on aspects such as the data characteristics (e.g., one-dimensional, multidimensional), workload (e.g., read-heavy, write-heavy), hardware (e.g., distributed, in-network, RDMA, GPU) and performance requirement (e.g., latency, throughput, storage size). Each index has its own set of advantages and optimizations.

2.3.2 Indexing Trajectories

We focus on the indexing structures for trajectories containing at least three dimensions (x, y, t). There are four main approaches to index trajectories [51, 10]:

- *The multidimensional approach* uses any multidimensional index and treats time as an additional dimension. Some notable works include the 3D R-tree [186], STR-tree [146], Grids [169], TB-Tree [146], Octree [103], and k-d tree [26].
- *The dimension reduction approach* uses space-filling curves (e.g., Z-ordered curve) to hash the spatiotemporal data into a one-dimensional key for indexing. The key can then be indexed with any one-dimensional index. Some notable works include CoPST [97], GeoMesa [74], GeoSOT [149], and JUST [121].

- *The multi-version approach* stores a spatial snapshot (with a spatial index) for each timestamp or time slice. Common nodes in the spatial indexes can be shared between timestamps to save storage costs. A query first finds the intersecting time slices before executing a spatial range query in each time slice. Some notation works include MR-tree [201], HR-tree [138, 183], MV3R-tree [184], and ST-Hadoop [13].

- *The spatial-first approach* builds a spatial index (e.g., R-Tree [91], quadtree [155]) and stores a temporal index (any one-dimensional index such as the B+tree [89]) at the leaves of the spatial index. The query first checks the spatial closeness before filtering by temporal closeness. The idea behind this design is that the change in spatial dimension is usually limited while the temporal dimension grows continuously [135]. This is one the most popular index approaches for trajectories, and some notable works include SETI [37], MTSB-tree [219], CSE-tree [192], PRADASE [125], TrajStore [44], Trajectory ST-Index [23], SHAHED [65], DITA [166], PITS [32], and GeoAnalytics [195].

We classify the indexes in the four distinct approaches into three main categories: one-dimensional, spatial, and multi-dimensional. One-dimensional indexes store keys in the dimension reduction approach and time in the spatial-first approach. Spatial indexes are used in the spatial-first and multi-version approaches (spatial snapshot). Multi-dimensional indexes serve as the primary index for the multidimensional approach, and the notable works listed above are considered multidimensional trajectory indexes. In the subsequent section, we review representative index structures for each index category, shedding light on their structure and optimizations.

2.3.3 One-dimensional Indexes

One-dimensional indexes store hashed keys from space-filling curves and timestamps in spatial index leaf nodes. There is a range of one-dimensional indexes designed for different access patterns. For example, hash tables are efficient for point lookup, and bloom filters excel at determining whether data exists. The access pattern for hashed keys and timestamps requires access to continuous intervals (or ranges) and is often updated. Therefore, the index structure

should efficiently handle range queries (e.g., find all timestamps between t_1 and t_2) and updates (e.g., append timestamps from newly inserted records and delete timestamps from expired records).

Indexes designed for range queries keep the data sorted internally for efficient query processing. Only one lookup is needed to find the first key that satisfies the range's lower limit. We can then linearly scan all records up to the range's upper limit. For example, we find the first timestamp greater than or equal to t_1 before scanning timestamps until it is greater than t_2. This process is more efficient than looking up each key in the range. We now introduce some common index structures for range queries.

B-tree & B+tree: B-trees are one of the most widely used database indexes for range queries. It is built on top of an array of n key-value pairs sorted over keys and supports searching any key within a $O(\log n)$ time complexity. The B-tree is designed as a balanced tree with all leaf nodes located at the same depth. The original B-tree stores the records in internal and leaf nodes.

Currently, the most common design of the B-tree is the B+tree [88], where the leaf nodes hold all records and internal nodes are purely for indexing purposes. The leaf nodes are connected via pointers (structurally similar to a linked list) to provide faster sequential access, making it particularly efficient for range queries. However, this comes at the cost of having duplicate data at the internal nodes (more on this in Chapter 6).

We describe the operations supported by B+trees since it is a de facto standard general-purpose index [62], and is considered the baseline in most index comparisons.

- *Search or Lookup:* The search finds the node that contains the query key k. The process starts from the root node and iteratively traverses down the tree. For each internal node comprising `(keys, pointer)` pairs, the search process checks if k is less than a key in the node. If the condition is true, the search moves to the left child node (via a pointer). Otherwise, it traverses down to the right child node. Once the leaf node is found, the matching key is found using search algorithms. Typically, binary search is used.

- *Range Search:* The process follows the lookup process. The only difference is that the query key is the lower limit k_l of the search range $[k_l, k_u]$. Once the leaf node is found, we search for the first key larger or equal to k_l. Then, we linearly scan each key, traversing nodes in a linked list fashion and stopping once the key is larger than k_u.

- *Insertion:* To insert a new key-value pair (k, v), the tree looks up the leaf node with the key range containing k. The key k is then inserted (with the value) in sorted order. The process is complete if the node has space to accommodate the newly inserted record. Otherwise, the leaf node is split in half, and the first key of the second node is inserted into the parent node. This may have a cascading effect in which the split propagates upwards until the root node. A new level is created if the root node is split, increasing the tree's height.

- *Deletion:* To delete a key-value pair (k, v), we lookup the k in the B+tree and remove the record. The process is complete if the number of records does not fall below the minimum occupancy (half-full). Otherwise, the tree must re-balance itself by redistributing keys or merging nodes. It first attempts to redistribute keys by borrowing keys from neighboring nodes. If redistribution causes neighboring nodes to underflow, the nodes are merged, and the reference to the merged nodes is deleted in the parent node. This change also has a cascading effect which may decrease the tree's height.

- *Record update:* To update a key-value pair (k, v) with a new value v_{new}, we simply lookup the k in the tree and replace v with v_{new}.

Throughout the years, research has been done to improve B+tree for modern CPUs. Some examples include the CSB+tree [150], which optimizes the cache performance of B+trees by grouping all child nodes in an array and storing pointers to the first child node. The BW-tree [117] allows concurrent access using atomic compare and swap (CAS) instructions with a latch-free design. B+trees have also been optimized for different hardware and workloads, including the Bz-tree [18] for persistent memory, the distributed B-tree [6] for distributed processing, and the IMtree [162] for streaming joins. Recently, with the rise in ML, there have

been an increasing number of studies to improve the B+tree structure with knowledge about the data, known as learned indexing. We shall explore learned indexing further in Section 2.4.

Tries & Radix Trees: Tries or prefix trees efficiently index keys from a fixed set of characters. Unlike B-trees, tries do not store the entire key in each node. Instead, they break down each key into components or characters distributed across different nodes. Tries are traversed depth-first, where each node includes one character. The path from the root to a leaf forms the key. A compact trie that allows each node to contain multiple characters is known as a radix tree.

The state-of-the-art trie-type structure that supports range queries is ART [113], a read-only radix tree that uses adaptive node structure with path compression and lazy compression to achieve space efficiency. Later works optimize tires by reducing the height [30] and incorporating range filters [211]. Tries are also combined with B-trees to support longer keys [127].

Queues: Queues are linear structures that obey a FIFO (First In, First Out) principle, where the first element inserted into the queue is also the first one to be removed. The queue insertion is known as enqueue, and the deletion is known as dequeue. The most common queue design is the circular queue. Searching a queue is the same as searching an array (or linked list, depending on the queue construction). Hence, it does not provide any search speed up. Queues are used to update heavy workloads such as streaming. We will discuss this in detail in Chapter 5.

LSM-Tree: Another update-heavy optimized index is the Log-Structure Merge Tree (LSM-tree) which buffers inserted data. Specifically, LSM-tree is designed for insert-heavy workloads. LSM-trees comprise exponentially larger levels, where each level is k times larger than the previous level [19]. The keys are inserted into the first (smallest) level in a sorted order. Once k nodes in the first level are full, they are sort-merged into the next level. This process has a cascading effect until the last level needs to be merged to form a new level. The design is optimized for lower levels to be in memory and higher levels in the disk, where the higher level benefits most from the sequential writes of LSM-trees. Lookup in the LSM-tree tries to find the data at the lowest level (typically with a bloom filter). If the data is not found, it moves to subsequent levels. Furthermore, LSM-tree allows for duplicate keys. Therefore, range queries are specifically slow since all the levels are searched, and the duplicate records are filtered.

2.3.4 Spatial Indexes

Spatial indexes are used when indexing approaches (the multi-version and spatial-first approach) disassemble trajectories into their spatial and temporal components. The key to efficient spatial queries is to store geographically proximate spatial data close together in the storage medium (e.g., memory or disk), often known as data locality. Thereby, the query only accesses the relevant data, which reduces the I/O cost (disk) and pointer lookup (memory) [7]. One-dimensional data, such as timestamps, numerical values, and strings, easily achieve data locality with sorting. However, data locality is much harder for spatial data (two dimensions or higher) as it has no inherent order and needs to be organized into one-dimensional storage layouts. Therefore, how to order the data to achieve data locality is a crucial design consideration for spatial indexes.

Two types of indexing strategies arose to achieve data locality: space-oriented and object-oriented indexing. Space-oriented indexing generates nodes by spatial decomposition, splitting the space of the dataset into smaller regions. On the other hand, object-oriented indexing generates nodes solely based on the spatial extent of the objects. The spatial extent can be the coordinates of points or bounding boxes of lines and polygons. The following paragraphs introduce some common space-oriented and object-oriented indexes.

Grids: The most intuitive space-oriented index is a uniform grid, similar to what is seen on maps [189]. The space is split into equally sized cells, each containing references to overlapping objects. If the object overlaps multiple cells, its reference is also duplicated.

To search using a uniform grid, the query object first finds the overlapping grid cell(s). All objects referenced in the cell(s) are compared with the query object. Each grid can accelerate the comparison by building a local index (e.g., vantage tree [69]). To insert an object, the object's reference is inserted into the cells it overlaps.

The uniform grid performs worse when objects' spatial distribution is non-uniform, also known as data skews. This is quite common in the real world; rural areas like deserts are less populated than urban city centers. More concentrated cells can have 3× more objects than the average

occupancy [189]. Indexes such as the grid file [140] and field-tree [75] have been developed for non-uniform data distribution. Briefly speaking, the grid file adjusts itself to the density of the data by allowing non-equal cell divisions, and the field tree uses multiple grids with different resolutions and displacements.

k-d trees: k-d trees are space-oriented binary-tree style indexes that break the space into axis-aligned hyperplanes [26]. Although the k-d tree can store lines and polygons via centroids of their bounding boxes [132] or by storing bounding boxes as higher dimensional points [153], it is designed to index points, where k represents the dimension of the points (e.g., for the spatial case $k = 2$). The root of the tree represents the entire spatial region. The first level divides the region into two axis-aligned hyperplanes. Each internal node represents an axis-aligned hyperplane. The splitting is done recursively based on a specified axis where the left subtree contains all points less than the node in the specified axis, and the right subtree holds all points greater. In terms of spatial data, the space is split in the x-direction on odd levels and y-direction on even levels.

A lookup in a k-d tree starts at the root until finding the point or reaching the leaf. During range search, each node checks whether their two subtrees overlap the query range before recursively applying the check to the subtree that satisfies it. In spatial queries, a lookup is considered a range search with a range of one. For insertions, each newly inserted point will result in a new node (property inherited from the binary tree). To insert a point, we traverse the tree to the leaf node that contains it before splitting the leaf nodes' region by the point.

The main disadvantage of the original k-d tree is that the structure of the tree depends on the order in which points are inserted (due to the nature of the splitting algorithm). In extreme cases, the tree can have n levels for indexing n points [189]. Newer k-d trees, such as the adaptive k-d tree [27], were proposed to compensate for the disadvantage. The idea was to generate more equal splits by ensuring the number of points on each side is (roughly) equal. A similar idea was used in the bintree [176], where equal-sized regions were used instead of an equal number of points. Another modification known as the k-d b-tree [152] adds B-tree characteristics to k-d trees that allow multiple points to be stored in the leaf node.

Quadtrees: Quadtrees are space-oriented indexes that recursively subdivide the space into four quadrants [155]. Similar to the k-d tree, the root consists of the entire area. However, each internal node in a quadtree represents four subspaces rather than two. The four subspaces or cells are typically called the northwest, southwest, southeast, and northeast regions.

The structure of the quadtree depends on the data type. For point data, the point quadtree [73], similar to the k-d tree, stores a point that splits the region as a node. Since each region is split into four, and some regions may be unpopulated. The search and insertion algorithms follow that of the k-d tree. The disadvantage of the point quadtree also follows that of the k-d tree, where the tree's structure depends on the insertion order. The PR quadtree [155, 141] mitigates this disadvantage by splitting the regions such that each leaf contains no more than x points (for a binary split: $x = 1$). Note that the PR quadtree is not balanced since denser regions may require more splits.

For line and polygon data, a region quadtree [155] recursively divides the regions until each region is either completely inside or outside the polygon. The recursion can also early stop by using a maximum depth threshold. The final quadtree will represent a raster approximation of the polygon. Alternatively, edge quadtrees [168] store lines instead of points where the splitting stops when each region contains one line segment or when the maximum depth hits the threshold. The problem with edge quadtrees is that the edges close to the cell vertices may be infinitely split until reaching the maximum depth. The PM quadtree [156] prevents such situations by not subdividing the cells if the segment intersects a vertex.

R-trees: R-trees are object-oriented indexes based on a hierarchy of minimum bounding boxes (MBB) [91]. It is one of the most popular indexes in commercial databases [51]. The R-tree is a multidimensional extension of the B+tree, which shares many properties. They are both balanced trees, and they have a notion of fullness. The occupancy in each node (except the root) is always between 50% and 100%. Nodes that are full due to insertions are split in half. Similarly, empty nodes from deletions are re-balanced with their neighboring nodes. However, the data layout differs from the B+tree to accommodate higher dimensional objects. Each object (point, line, or polygon) is represented as a minimum bounding box stored in

the leaf node (e.g., as a pair of `(mbb, object_id)`). The internal nodes store the MBB that encompasses all its children and points to the child node with a pointer in the form of `(mbb, ptr)`. One advantage of R-trees is that the objects are stored as a whole instead of fragmented objects like quadtrees.

Searching the R-tree is based on using MBB as keys. The process is similar to a B+tree search. It starts at the root node and compares the query range with the MBB. If they overlap, the node is visited, and the process recurses to the leaf node. Multiple leaf nodes can be found since the MBB of nodes can overlap. Each object within the leaf nodes is then compared with the query range. The insertion process is also similar to that of the B+tree, where objects are inserted into the leaf nodes if they meet the occupancy constraint. Otherwise, the node is split, and the change may propagate upwards where the parent node's MBB is adjusted. A similar idea applies to the deletion process.

There are a lot of variants of the R-tree, most notably the R*-tree [25], which seeks to remove node splits through shuffling of objects (remove and reinsert), and the R+-tree [161], which seeks to minimize node access by reducing overlaps between MBBs, having fuller nodes and duplicating references to objects. Much research has been done on R-trees, such that there is a family of R-trees. One example is the STR-tree [116], which adds a sort-tile-recursive algorithm for bulk loading. This allows for multiple objects to be inserted into a R-tree at once. R-tree's ability to be easily extended to any dimension makes it popular for trajectory and multidimensional indexes, which we will review in the next section.

2.3.5 Multidimensional & Trajectory Indexes

Multidimensional indexes can be used to index trajectories by treating time as an additional dimension. This can be implemented by adding a dimension to spatial indexes (the multidimensional approach). For example, for three-dimensional trajectory data (x, y, t), we can extend the R-tree to a 3D R-tree [186], a quadtree into an octree [103], or a k-d tree with $k > 2$. Time is mapped on the z-axis, where the height represents the object's lifetime. The disadvantage of this method is due to the characteristics of trajectory updates. Although updates add new

spatial and temporal points, spatial data is usually confirmed within fixed regions (e.g., within the city's boundary). On the other hand, the temporal dimension continuously grows with newer timestamps [144]. This introduces a lot of empty space, known as dead space in the index, especially when storing trajectories with a set of line segments [51]. Additionally, objects with a long duration will be stored as long rectangular prisms, generating many overlaps. This results in more node access during queries which decreases performance (especially for the 3D R-tree) [94]. Therefore, one key challenge for multidimensional indexes is how to cluster the data to take advantage of the characteristics of trajectories.

Multidimensional trajectory indexes apply techniques to achieve better query performance, such as adding a clustering algorithm to consider grouping by trajectories rather than spatial proximity, combining different types of spatial and one-dimensional indexes, and accounting for the underlying movement pattern (unconstrained, constrained, and network movements). Furthermore, more recent trajectory indexes also leverage distributed systems to process trajectories at a large scale. Some of the most popular platforms are Hadoop [17] and Spark [209], both based on the MapReduce [50] architecture. We present some common trajectory indexes below:

STR-tree & TB-tree: The Spatio-temporal R-tree (STR-tree) [146], not to be confused with the sort-tile-recursive R-tree with the same acronym, optimizes the 3D R-tree for trajectories with a better clustering algorithm. It uses a multidimensional approach similar to the 3D R-tree. The only difference between the two is the insertion and split algorithm. The idea behind the STR tree is to add partial trajectory preservation on top of spatial closeness by storing line segments belonging to the same trajectory together. Therefore, the insertion algorithm tries to insert the line segment close to the preceding line segment with the same trajectory ID. However, achieving trajectory preservation and spatial closeness is not always possible. A preservation parameter p was introduced to balance between the two [146], where lower values reduce trajectory perseverance and increase spatial closeness.

The trajectory bundle tree (TB-tree) [146] takes the trajectory perseverance property of STR-trees even further by storing segments from the same trajectory in a leaf node, also known

as bundling. If more than one leaf node is required to store the segments, the leaf nodes are connected via pointers similar to a doubly listed list. The insertion algorithm is adjusted such that if a leaf node is full, it creates a new leaf node (rather than splitting) to preserve the trajectory. The design excels at retrieving a complete trajectory with minimal node access. The main disadvantage is that spatially close line segments are stored in different leaves [144].

MR-tree & HR-tree: The Multiple R-Tree (MR-tree) [201] takes the multi-version approach by storing separate R-trees for each timestamp. The idea is to build an R-tree for each timestamp instead of augmenting a temporal dimension to multidimensional indexes [51]. One can simply build a B+tree on the timestamps and store an R-tree in each leaf node. However, this would result in an unpractical amount of storage. The MR-tree reduces storage overheads by sharing common objects between timestamps. Only objects with changed positions are created, and unchanged nodes are linked to different R-tree roots from different timestamps.

The Historical R-tree (HR-tree) [138] uses the same idea as MR-tree with a more concrete update algorithm. However, the HR-tree tends to degenerate to a single R-tree per timestamp due to its constraint on not allowing different versions of objects to be placed in the same node [183]. The HR+-tree breaks the constraint by allowing objects with small changes to remain in the node. R-trees in different timestamps can share the node instead of generating a new node for every small change. However, all three indexes still generate duplicate objects, which makes them large and perform quite poorly for temporal range queries (or time-interval queries) [37].

MV3R-Tree: The MV3R-Tree [184] is a multi-version approach that uses a "dual" index type structure by combining a multi-version R-Tree and a 3D R-tree, where both structures share the same leaf nodes. The idea is to perform temporal snapshot queries on the multi-version R-tree and temporal range queries on the 3D R-tree. However, this design means the insertion algorithm is quite complex as it essentially needs to balance two indexes. Therefore, for frequently updated objects (e.g., objects with gradual changes), the performance of MV3R-tree suffers [37].

SETI, MTSB-Tree & CSE-tree: The Scalable and Efficient Trajectory Index (SETI) [37] is a grid-based index designed for the characteristics of trajectories where spatial dimensions are usually confined and temporal dimensions continuously increase. The idea is to index the spatial dimension, which may be constant over time, before adding a dynamic index to index the timestamps in each spatial partition. SETI uses a grid index to partition the space. A segment that intersects multiple grid cells is cut at the boundary and inserted into the respective cells they overlap. Note that the timestamp at the split point is interpolated from the segment. The timestamps in each cell are indexed with an R*-tree.

The MTSB-tree [219] is a SETI variant that uses a Time-Split B-tree (TSB-tree) to index the time dimension in each cell. The only difference from SETI is that the query results will be ordered in time. The Compressed Start-End tree (CSE-tree) [192] is another variant of SETI which changes the temporal index depending on update frequency. It uses a probabilistic model to identify heavily updated data. A B+tree is used for indexing dynamic data, and a sorted dynamic array is used for (relatively) static data. The main drawback of SETI-based methods is the uniform grid used to partition the spatial dimension leads to poor performance with skewed data distribution.

Space-Filling Curves: The idea of space-filling curves is to map multidimensional space into a one-dimensional line to preserve the spatial locality. Theoretically, points in multidimensional space remain close when mapped onto the curve. The principles are the same regardless of the number of dimensions. Space-filling curves map the space into a uniform grid, then traverse each grid center without crossing itself. A unique curve number is stored as the ID of the cell, which is indexed with one-dimensional indexes such as the B+tree. Objects are stored under the cell(s) they intersect. The two most common types of space-filling curves are the Z-order and Hilbert curves due to their range query performance compared to others [189]. The Z-order or Morton curve [136] is based on the interleaving of binary coordinates, and the Hilbert curve [3] is based on the Hilbert-Peano curve. They differ only by how the adjacency and ordering of the cell are defined.

Distributed Indexes: Recently, there has been a surge of trajectories indexes in distributed systems for handling data at scale. Typically, these indexes are designed for specific platforms (e.g., Hadoop and Spark) where their main goal is to achieve scalability. A major design consideration is how to partition the data to achieve load balance. Therefore, many of these structures combine partitioning and indexing, differentiating them from traditional indexes. For example, ST-Hadoop [13] uses a multi-version approach (spatial index in each temporal slice). The time slice can be chosen such that each time slice has a similar number of points to achieve load balance. Any spatial index can be applied to the data in the time slices. A full review of distributed trajectory indexes can be found here [188]. We will discuss more about specific distributed indexes for joins in Chapter 3.

2.4 Learned Indexes

2.4.1 Introduction to AI & DB

The concept of learned indexes falls in the greater category of artificial intelligence for databases (AI4DB). The idea behind AI4DB is to incorporate machine learning techniques and algorithms to optimize database components, such as indexing, cardinality estimates, and database tuning [222]. This has been an active area of research for the past few years, alongside using database techniques to improve AI models (DB4AI) [118]. Together, these two areas show the recent trend of adopting AI into various fields and how AI impacts systems research.

We focus on AI4DB, where AI techniques are introduced to improve empirical methodologies and human tuning in traditional database design [118]. One advantage of AI techniques is that they can explore a larger design space much faster than humans. This allows them to find patterns that have yet to be found by humans, which makes them ideal for addressing hard problems in databases. Current AI4DB techniques can broken into five categories, which we briefly introduce below. A full review can be found in [118, 220]. The following introduction lets the readers understand how learned indexes fit under AI4DB research.

1. *Learned Database Configurations:* AI models are used to automate the database configuration tuning process, such as database knob tuning, index recommender, and materialized view advisor. For example, in database knob tuning, numerous parameters in the database can be tuned for different workloads. Due to the number of parameters involved (which generates a high-dimensional parameter space), the system typically uses complex models, such as deep and reinforcement learning, to select optimal configurations under various workloads [212].

2. *Learned Query Optimization:* As mentioned in Section 2.2.1, the best execution plan that minimizes the required resources is found before executing the query. AI models are alternatives to heuristics rules, especially for harder problems where heuristics fail. The models can be applied to specific parts of the optimizer, such as cost estimation [173], join order selection [130], cardinality estimation [197], or used as an end-to-end optimizer [129].

3. *Learned Database Design:* AI techniques optimize existing data structures and support self-designing data structures that adapt to different workloads [60]. Learned indexes fall into this category as AI-optimized data structures, which we will discuss in detail in Section 2.4.2.

4. *Learned Database Monitoring:* The purpose of monitoring database activities is to diagnose failures and ensure performance during runtime. For example, monitoring database performance is done by predicting the query latency, which is highly dependent on many factors such as execution plan, data distribution, and resources available. Deep learning can be used to capture the interactions between these factors [131].

5. *Learned Database Security:* Database security protocols prevent vulnerabilities such as unauthorized data observation, modification, and unavailability [28]. The goal of AI models is to help prevent and detect unknown security vulnerabilities.

Lastly, we also want to mention that there are full AI4DB database designs, known as instance-based databases. Most notably, SageDB [110, 55] combines many of the learned components from above into an entire system. Recently, PilotScope [221] has been proposed to act as a

driver to allow AI4DB techniques (some mentioned above, such as deep learning configuration tuning [212]) to be more easily integrated into databases.

2.4.2 Introduction to Learned Indexes

The idea of learned indexes is to use machine learning as a prediction technique to quickly narrow the search range in an index. The original concept was introduced by Kraska et al. [111], where they applied the idea of learned indexes to an in-memory B+tree. Specifically, the internal structure of a B+tree can be viewed as a model that predicts the position of the data given its key. However, the B+tree "model" is generic and does not exploit the underlying data distribution. For example, the lookup time of B+trees is $O(\log n)$ for a n-sized dataset regardless of whether the keys are consecutive or random integers. Therefore, a machine learning model, learning the empirical cumulative distribution function (CDF) of the indexed data, can instead provide a better prediction. In our example above, the lookup for consecutive integers can be found in $O(1)$ time compared to the $O(\log n)$ time of the B+tree.

We can formalize this learned index approach by considering sorted data of integers $K = \{k_0, k_1, ..., k_{n-1}\}$ (e.g., an array) that is indexed with a structure I (e.g., a B+tree). We can view I as a model or function that maps an integer key $k \in \mathbb{Z}$ to a search bound $[i_{low}, i_{high}]$, where $i \in \mathbb{Z}_{\geq 0}$):

$$I(k) = [i_{low}, i_{high}] \tag{2.1}$$

For I to be a valid index structure, any key k must be able to find a bound that contains its lower bound $lb(k)$. The lower bound of k is the smallest key in K that is greater than equal to k. Note that if k is larger than the largest key in the dataset ($\max K$), then the lower bound is $n-1$. This is equivalent to the `lower_bound()` function in the C++ standard library. Formally, the lower bound is defined as the following [128]:

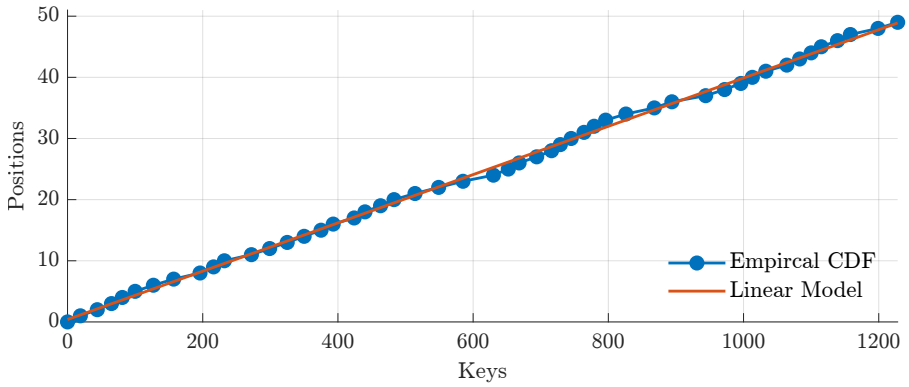


Figure 2.1: The database distribution of a sorted array of keys is represented as a cumulative distribution function (CDF).

$$lb(k) = \begin{cases} i \in \mathbb{Z}_{\geq 0} \mid K_i \geq k \wedge (\nexists j | j < i \wedge K_j \geq k), & \text{if } k < \max K \\ n-1, & \text{else} \end{cases} \tag{2.2}$$

We can formulate the equation for a valid index structure I to be:

$$I(k) = [i_{low}, i_{high}], \forall k | i_{low} \leq lb(k) \leq i_{high} \tag{2.3}$$

This formulation is valid for any range index. For example, if we treat I as a B+tree, the search range (i_{low}, i_{high}) would be the range of the leaf node that contains the lower bound of the key. To find the exact position, a last-mile search within the search bound would be carried out (e.g., a binary search in the case of a B+tree). A B+tree can only reduce the search bound by reducing its fanout. This generates a much taller tree, which increases traversal costs. The structure would resemble a binary search tree as the fanout is reduced.

The idea of learned indexes is to reduce this last-mile search bound by capturing the data distribution. The data distribution is approximated with a cumulative distribution of the keys, where the x-axis is the key, and the y-axis is the position of the key in the dataset shown in Figure 2.1. We can term the cumulative distribution function (CDF) as the mapping function from keys to positions [128, 21]. In terms of the statistical definition, the CDF of key k returns the probability of keys that is less than k*. Therefore, the objective of learned indexes is to

*for a 1:1 definition, the CDF in Figure 2.1 needs to be normalized.

model the CDF. As with all models, there is error $\delta()$, shown as the difference between the linear model and the empirical CDF in Figure 2.1. The error is the learned index's last mile search bound. To search key k using a learned index I_L, the model predicts the position i_{pred} of k and returns a search bound based on the model error $\delta(k)$. We can formalize the search bound as the following:

$$I_L(k) = [i_{pred} - \delta(k), i_{pred} + \delta(k)], \forall k | i_{pred} - \delta(k) \leq lb(k) \leq i_{pred} + \delta(k) \tag{2.4}$$

If the maximum error $\Delta = \max \delta(k), \forall k \in K$ is known, then the exact position of key k is bounded by:

$$I_L(k) = [i_{pred} - \Delta, i_{pred} + \Delta], \forall k | i_{pred} - \Delta \leq lb(k) \leq i_{pred} + \Delta \tag{2.5}$$

The obvious and most intuitive way to improve learned indexes performance is to reduce the error. The previous example where the data is consecutive integer keys is the extreme case where the distribution is easy to learn. We can learn a perfect model (e.g., linear regression) with no error. Hence the index can achieve a lookup time of $O(1)$. However, as we will see in the next section, this is not true for most key distributions.

2.4.3 Design of Learned Indexes

The design of learned indexes is not as simple as finding the most complex model that minimizes the error when approximating the CDF. Although the CDF can be easily captured from a machine learning point of view, a neural network [111] performs worse than a B+Tree due to two factors. First, if we increase the complexity of the model, the prediction cost also increases. Second, a single complex model is able to capture the general shape of the CDF well. However, it is unable to model the irregularities at the granular scale. Hence, the error at the individual record level is large, which results in a much more expensive last-mile search. Therefore, the general structure of learned indexes uses multiple less complex models to divide and conquer

the approximation of the CDF.

The first of these is the Recursive Model Index (RMI) [111], which supports a range of models from linear to non-linear models. The idea is to build a hierarchy of models to refine the search. Starting from the root, the model first takes a prediction for a specific key. The initial prediction serves to give the general location in the CDF. Once the general location is known, a child model responsible for that area refines the prediction until reaching the leaf level. The leaf level is responsible for the most granular level and gives a final correction to the prediction before carrying out a last-mile search.

Almost all newly proposed learned indexes adopt this strategy. However, they differ in implementation and design considerations (such as minimizing storage cost, allowing updates, and supporting concurrency). For example, the RMI does not use a search to correct prediction at each level, while other learned indexes such as the PGM-index [72] apply a correction at each level. Additionally, RMI is designed for static data and does not support structural modifications such as insertions. To support inserts, the FITing-tree [78] builds itself from the bottom-up and adds a buffer to each leaf node. We will briefly introduce the design considerations and refer readers to [174] for a detailed evaluation.

- *Lookup design:* The lookup design determines the type of models used to fit the data and the correction process
  - *Model design:* The main types of models are linear and non-linear models. Linear models are the most widely used for their effectiveness on real-world datasets [108, 128] and low retrain cost. A few indexes use non-linear models, including RMI, which supports a cubic model, and LIPP [198], which uses a polynomial model. Note that none of the non-linear models are too complex. Additionally, a hybrid model can also be used by combining linear and non-linear models or even using a B+tree for the internal structure [78].
  - *Correction process:* The correction process determines how the prediction position is corrected. The error can be corrected at the leaf node or at each level (as in

the example above). Typically, indexes that correct error at each level do so to reduce the height of the tree and provide theoretical guarantees (e.g., PGM-index [72], XIndex [180], FINEdex [119], FLIRT (Chapter 5), and SWIX (Chapter 6)). Alternatively, there may not be any error in the case of LIPP [198]. The correction process itself uses linear, binary, or exponential search. Linear search is reserved for small error ranges (details in Chapter 5), while binary search is used for larger error ranges. Exponential search is effective when the actual position is close to its predicted position (details in Chapter 6).

- *Insertion strategy design:* Since learned indexes are based on a trained model and need to keep the data sorted, insertions require shifting of the data, which reduces the model's prediction accuracy. More data also means that the index structure needs to be adjusted (or re-balanced in terms of B+trees). However, unlike traditional indexes, modifications to the learned index mean that the model needs to be retrained, which is expensive (time-wise). Therefore, the design of insertion strategies determines how the learned index deals with insertion and structural modifications.

  - *Supporting inserts:* There are two ways for learned indexes to support insertions. The first is to use a buffer that can be periodically merged (e.g., when the buffer is full) with the existing index. The buffer can be placed at the index, node, or data level. For example, the dynamic PGM-index uses an LSM-tree style structure which buffers all data at the index level. FITing-tree [78] and XIndex [180] all have buffers in each leaf node, while FINEdex [119] uses a list-of-array style B-tree to buffer insertions at the data level. The second method is to reserve gaps in each node by pre-allocates spaces during model training for new records (e.g., ALEX [56]). Note that indexes can use a hybrid of the two, as we will see with SWIX.

  - *Structural Modification Triggers:* Indexes use triggers to determine when to adjust their structure (similar to the fullness criteria of the B+tree). The goal of learned indexes is to reduce the number of structural modifications. The triggers are based on fullness, error, and cost thresholds. The fullness threshold is seen in practically every

index to prevent the buffer or node from getting too large. Error-based thresholds are used in indexes with a max error threshold (predefined or dynamically generated) to ensure prediction accuracy. Cost-based thresholds, particularly ALEX, take another approach by estimating the cost of operations in a node (lookup and insert time) and ensuring the cost is lower than a threshold. Note that a single index can use multiple triggers.

- *Supporting deletes:* Deletes can trigger structural modifications similar to insertions. For learned indexes, deletes are triggered with the lower bound fullness threshold, which determines the least amount of keys that can exist in a node (e.g., ALEX [56], XIndex [180], SWIX). The idea is similar to that of the B+tree. Regarding the actual delete operation, records are typically flagged in the node and are only erased if they are in the buffer. We can also group insertion and deletion into a single component: *update strategy design.*

- *Bulk-load design:* The bulk load algorithm can be split into top-down and bottom-up (as with traditional indexes). For top-down approaches, all data is first placed in the root node. Then the data is split into multiple child nodes until reaching the leaf level. For each internal node, a model is trained based on key-to-child mapping. On the other hand, the bottom-up approach splits the data into leaf nodes. Each leaf's first and/or last keys are used for the level above. The main difference between traditional indexes and learned indexes is the splitting criteria. The splitting criteria for learned indexes take into account the prediction accuracy. Some splitting method considers load balance, such as even splitting (split the nodes evenly while ensuring the error in each node is lower than a threshold [180]). Other splitting criteria, such as error-based and cost-based methods, purely split nodes based on prediction accuracy. Specifically, error-based methods incrementally add records to the node until the error threshold is reached [78, 72], while cost-based methods estimate the cost of operations in each node and compare that with a cost threshold [56].

2.4.4 Baseline Learned Indexes

Currently, there are over 46 learned index designs for one-dimensional data and 27 multidimensional learned indexes. We present baseline learned indexes used in Chapter 5 & 6.

The PGM-index [72]: The PGM-index is a hierarchical learned index where each node uses an error-bounded linear regression. It is the first learned index to provide theoretical guarantees. It uses a bottom-up bulk-load design, generating the leaf nodes from the data using an optimal Piecewise Linear Approximation algorithm based on a user-defined error threshold. The algorithm ensures the minimal number of nodes while guaranteeing the error thresholds. The internal nodes are built recursively with the same algorithm. The PGM-index traverses from the root node to the leaf node during the search. Since each node has an error, the prediction is corrected with a binary search at each level. To support updates, the dynamic PGM-index is designed with an LSM-tree style structure comprising a set of smaller PGM-indexes. Newly inserted records are first placed into a buffer, once the buffer is full it is trained to form a PGM-index. The smaller indexes are also merged to form larger trees in the process.

ALEX [56]: ALEX is an updatable hierarchical learned index based on cost models. Unlike the PGM-index, it is an unbalanced tree where leaf nodes can be at different depths. The highlight of ALEX is the cost model that estimates the cost of latency of search and insertion operations of each node for controlling structural modifications and the bulk load algorithm. ALEX uses a top-down bulk-load approach. Starting from the root, ALEX attempts to split the key range in n child nodes. n is chosen to minimize the estimated cost in each node with a fanout tree. Note that n is a power of 2 as the fanout tree is based on a binary tree. The top-down approach ensures that only the leaf nodes have errors. ALEX uses a linear regression for a lookup and a gapped array to accommodate insertions. Unlike the dynamic PGM-index, ALEX allows the nodes to expand and split with more insertions. The structural modification triggers use the same cost model in the bulk load.

XIndex [180] & FINEdex [119]: XIndex and FINEdex are concurrent learned indexes. XIndex is built on top of the RMI structure with buffers in each node to store newly inserted

keys. FINEdex improves upon XIndex using a heuristically built shallower structure and data level buffers in the leaf node for inserted keys. Compared to other learned indexes, the most important difference is the concurrency design, which prevents race conditions between threads when modifying the index structures during insertions. XIndex uses a temporary buffer to accommodate inserted keys from other threads when one thread splits and retrains the full buffer. FINEdex, on the other hand, allows threads to concurrently update parts of the node using fine-grained locks. Specifically, it only locks the full data level buffers, allowing threads to access other buffers in the leaf node.

Lastly, we want to mention that we do not use multidimensional learned indexes as baselines since they are designed to handle point datasets and do not support trajectory data or trajectory-based queries. The only multidimensional learned index that supports non-point geometries is GLIN [190], which only supports spatial data and topology queries.

Chapter 3

Distributed Trajectory Distance Join

3.1 Overview

In this chapter, we present our distributed framework for efficiently answering distance joins from a global perspective, where we seek to improve the performance across multiple machines. The distance join is important in applications in many fields. A simple example is identifying near misses in maritime applications where vessels near (spatially and temporally) each other must be identified to reduce the number of close encounters. A distance join on these vessels can identify the hotspots where near misses occur and help maritime authorities plan avoidance measures. We can give another example for higher-dimension trajectories where an astrophysicist may be interested in capturing extraterrestrial objects that pass by each other.

Current research on trajectory joins focuses on the kNN join and similarity join. The kNN join seeks to find the k-closest trajectories from dataset Q for each trajectory in dataset P given a spatiotemporal distance criterion. Unlike the distance joins, it is unable to return all of the objects within a spatiotemporal distance. The similarity join, on the other hand, focuses on finding trajectories based on how similar they are to each other. Therefore, the shape of the trajectories must be taken into account rather than just joining based on distance.

Distributed systems have become more popular for enabling timely analysis and querying of

extremely large datasets. In recent years, extensive research has been made to handle large spatial datasets using distributed systems [64, 205, 206, 200, 181]. While designed to perform spatial operations on a large scale, none of them explicitly supports trajectories. Those that handle spatiotemporal datasets simply store trajectories as points, ignoring the spatial continuity of trajectories [145]. Therefore, queries about the trajectories' movement in between sample points cannot be answered. Recently, a number of distributed platforms [216, 165, 11, 22] have been designed specifically for trajectories. However, these systems do not currently support distance join queries.

Given the shortcomings of existing systems, we developed a distributed approach on Spark to answer distance join queries for large trajectory datasets. We propose a distance join framework that stores trajectories as a sequence of line segments before presenting three variants of the distance join for different applications.

Furthermore, we focus on how to efficiently answer distance joins for large volumes of data whilst reducing the impact of updates on data partitioning and query efficiency. For a join, updates also have to account for changes in the join predicate (in addition to new data) since the join cardinality heavily impacts the partition strategy. We use a uniform grid partitioning strategy and local grid index that is fast to update for incoming data. To account for the join predicate, we provide a search space method that minimizes the number of duplicates in each partition by correlating the grid size with the distance threshold (spatial join predicate) to reduce the number of intersecting grid cells. The in-partition local grid index, built based on the spatial and temporal predicate, improves the local join performance by allowing true-positive results to be outputted directly without the need for expensive distance computations.

3.2 Drawbacks of Existing Works

As mentioned in Chapter 1 and Section 2.3.5, the increasing volume and real-time requirements of trajectories render centralized single-machine systems inadequate. Distributed systems have emerged to improve the performance of large-scale data processing. Hadoop [17] and Spark [209]

(both based on MapReduce [50]) have risen to become one of the most popular platforms for spatial & spatiotemporal query processing. In this section, we provide an overview of existing join methods based on Hadoop and Spark and why they cannot meet our needs. Specifically, we first review existing join algorithms before examining Hadoop and Spark-based join algorithms.

3.2.1 Trajectory Joins Algorithms

Trajectory join algorithms can be categorized into three main categories: nearest-neighbor-based (kNN), similarity-based, and distance-based. Substantial research has been done on the first two categories. kNN joins return the k-closest objects to a query position. It can also find k-closest trajectories to a given trajectory [71]. The difference between the kNN and distance join is that the distance used for a kNN join is relative, and it does not return all objects within the distance. Therefore, the kNN join cannot, at least efficiently, answer a distance join.

Similarity joins aim to use a similarity measure to capture the shape similarity between trajectories. Distance may still play a part in the join predicate. However, the level of impact depends on the specific similarity measure since the main focus of similarity joins is not distance. There has been a lot of work [53, 163, 175, 47, 177] on designing similarity measures for trajectory comparisons. For example, [53] uses the Fréchet distance to consider both shape similarity and distance. Since the shape is always considered in the similarity measure, the similarity join predicates cannot, at least efficiently, answer pure distance-based joins.

Additionally, similarity measures have been proposed to improve other aspects of the join, such as removing the need for a time window [163], optimizing for not well-aligned trajectories [175], and incorporating similarity into the temporal dimension [47]. Another interesting direction is looking at sub-trajectories [177]. The idea is to find potential matches between portions of the trajectory since it may be difficult to use one measure to capture the entire trajectory. This inspires our method, which compares line segments instead of the entire trajectory.

Lastly, we review research on distance-based joins. These methods [151, 164] mainly focus on joins between trajectories and objects, which are ideal for applications such as capturing the

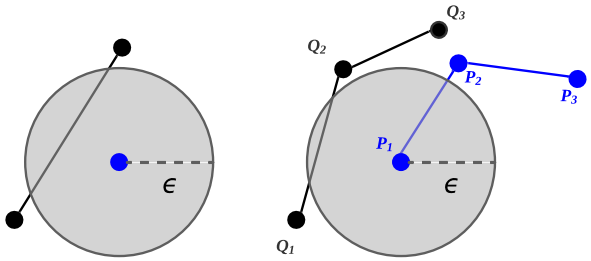


Figure 3.1: *Right:* False positives caused by the query radius not intersecting with endpoints. *Left:* Example when joining two trajectories P and Q.

movement of objects within a region [151] or near points [164]. Efficient distance-based joins between trajectories remain an unsolved challenge, which we attempt to tackle in this chapter.

3.2.2 MapReduce-based Trajectory Joins

We first briefly introduce Hadoop and Spark, which are based on the MapReduce architecture. The key advantage of MapReduce is that it simplifies the complexities of distributed systems into two key steps: the first "map" step divides the problem into sub-problems processed independently in parallel. The second step "reduces" or gathers the output from all the sub-problems. Hadoop is a disk-based framework based on MapReduce using a master-worker layout. Each cluster contains one master node, while the rest of the nodes are workers. Data is split and distributed across the nodes using the Hadoop Distributed File System (HDFS). MapReduce tasks are transferred to the nodes for parallel processing. Spark applies MapReduce to in-memory distributed computing by storing data in memory using a Resilient Distributed Dataset (RDD) abstraction. The advantage of Spark is that intermediate results can be kept in memory which reduces I/O and serialization from constant read-writes to disks.

We now review spatial systems prior to spatiotemporal & trajectory systems. [213] is the first Hadoop-based system that supports spatial joins. However, they did not include any indexing structure to accelerate the join process, which was improved by [8, 64]. Subsequent works [171, 205, 206, 181, 81] further accelerate the join performance by improving the join algorithm or introducing new features. Briefly speaking, the features include SQL support [171] and support for load balance when partitioning [181]. None of these systems supports spa-

tiotemporal datasets and queries (not to mention trajectories).

Existing multidimensional & spatiotemporal systems [200, 12, 95, 195] store datasets as multidimensional points. To perform a distance join between trajectories P and Q, these systems find all the points in Q that are within a spatial distance and time window from each point in P. Each point is treated as an independent point with no knowledge of preceding or succeeding points. As mentioned in Section 2.2.5, storing trajectories as points results in loss of information about the objects' movement in between points [145]. Figure 3.1 shows an example where the query radius may intersect the line segment but not the endpoints, resulting in false negatives from point-based systems. On the other hand, current trajectory-based systems [216, 165, 11, 22] does not support distance joins between trajectories. Furthermore, these systems store trajectories as segments, which we think is inefficient (details in Section 3.4).

3.3 Background

In this section, we provide concepts and notations used in the rest of the chapter.

3.3.1 Trajectory Data Model

Before introducing the query formulation, we first briefly recap the trajectory data model presented in Section 2.2.5. A trajectory is stored as a sequence of time-ordered spatiotemporal points. In 2D space, each spatiotemporal point p_i takes the form of (x_i, y_i, t_i), where x_i and y_i are positions along the x and y axis, and t_i is the timestamp of that position. Hence, mathematically, a trajectory T is defined as $\{p_1, \ldots, p_i, \ldots, p_n\}$ such that $\{t_1 ... < t_i ... < t_n\}$.

A segment is defined as the portion of a trajectory within the time boundary of the trajectory $[t_1, t_n]$. Formally, a segment between p_i and p_j is $s_{ij} = [p_i, p_j]$, where $t_i \geq t_1$, $t_j \leq t_n$ and $t_i < t_j$. A line segment is a segment between two consecutive points: $l_{i,i+1} = \overline{p_i p_{i+1}} = [p_i, p_{i+1}]$. The overline vector notation indicates that the movement between two endpoints is linear rather than two discrete endpoints.

3.3.2 Query Formulation

In general, a trajectory join takes two trajectory datasets and combines them based on a spatial and temporal condition [164]. Given two trajectory datasets P and Q, a distance join finds all pairs of trajectories based on a predicate $pred()$. The predicate depends on the type of object being compared. For the naive case where we treat the trajectories as points, the distance join is formulated as the following:

$$P \bowtie_{\text{pred}} Q = \{(p_i, q_j)\} \text{ where } \forall p_i \in P \text{ and } \forall q_j \in Q \text{ such that } pred(p_i, q_j) = \text{true} \tag{3.1}$$

Where, p_i is a point in P, q_j is a point in Q, and $pred(p_i, q_j)$ is based on the distance threshold ϵ. Two points are close if their distance is less than ϵ. A traditional distance join does not feature time. However, trajectories require a time dimension. There are multiple ways of including time into the distance join. One common method is to have a time window d. For points, this is simply checking to see if the difference between $t_i \in P$ and $t_j \in Q$ is less than d. If we extend the formulation for line segments:

$$P \bowtie_{\text{pred}} Q = \{(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}})\} \text{ where } \forall \overline{p_i p_{i+1}} \in P \text{ and } \forall \overline{q_j q_{j+1}} \in Q \text{ such that } pred(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}}) = \text{true} \tag{3.2}$$

$\overline{p_i p_{i+1}}$ is a line segment in P, $\overline{q_j q_{j+1}}$ is a line segment in Q, and $pred(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}})$ is the spatiotemporal relationship between the two line segments. The spatial component now checks if the shortest distance between the two line segments is less than ϵ. The spatial distance between two line segments $\overline{p_i p_{i+1}}$ and $\overline{q_j q_{j+1}}$ is defined as:

$$minDist(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}}) = \min\{|p, q|\} \text{ where } p \in \overline{p_i p_{i+1}} \text{ and } q \in \overline{q_j q_{j+1}} \tag{3.3}$$

Where, p is a point on line segment $\overline{p_i p_{i+1}}$, q is a point on line segment $\overline{q_j q_{j+1}}$, and $|p, q|$ is the distance between points p and q. We denote p_c as the point on $\overline{p_i p_{i+1}}$ closest to point q_c on $\overline{q_j q_{j+1}}$. This means that $|p_c, q_c| = minDist(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}})$. Any spatial distance measure can be used (e.g., Euclidean, Manhattan). For the temporal component, the most common method is to use time intersections. It is considered temporally close if $[t_i, t_{i+1}] \in P$ intersects with $[t_j, t_{j+1}] \in Q$. We can add a time window by extending the duration at the endpoints of one of the line segments (e.g., $[t_i, t_{i+1}] \rightarrow [t_i - d, t_{i+1} + d] \in P$), which leads us to the first type of spatiotemporal distance join.

Time window distance join: Given a distance threshold ϵ, and a time window d, return the line segments $\overline{q_j q_{j+1}}$ in Q such that $minDist(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}}) \leq \epsilon$ and $[t_i - d, t_{i+1} + d] \in P \cap [t_j, t_{j+1}] \in Q$ for each line segment $\overline{p_i p_{i+1}}$ in P.

The main drawback of the time window distance join is that the results may be spatially close but not necessarily temporally close since the temporal condition only checks for intersections and does not check whether the actual closest points (p_c, q_c) are temporally close. This is more likely to occur in line segments that have a long duration ($t_i << t_{i+1}$), which brings us to our next type of join.

Time windowed exact distance join: Given a distance threshold ϵ, and a time window d, return the line segments $\overline{q_j q_{j+1}}$ in Q such that $minDist(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}}) \leq \epsilon$ and $|t_{P_c} - t_{Q_c}| \leq d$ for each line segment $\overline{p_i p_{i+1}}$ in P.

However, there are applications where the time window d cannot be trivially defined. For example, in the case of maritime transportation where the temporal closeness value is different for vessels moving in port and in open water (due to the room they need for avoidance measurements). We propose another variant that compares the distance at the exact time instances, alleviating the need for a time window.

Time match exact distance join: Given a distance threshold ϵ, return the line segments $\overline{q_j q_{j+1}}$ in Q such that $minDistTime(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}}) \leq \epsilon$ for each line segment $\overline{p_i p_{i+1}}$ in P.

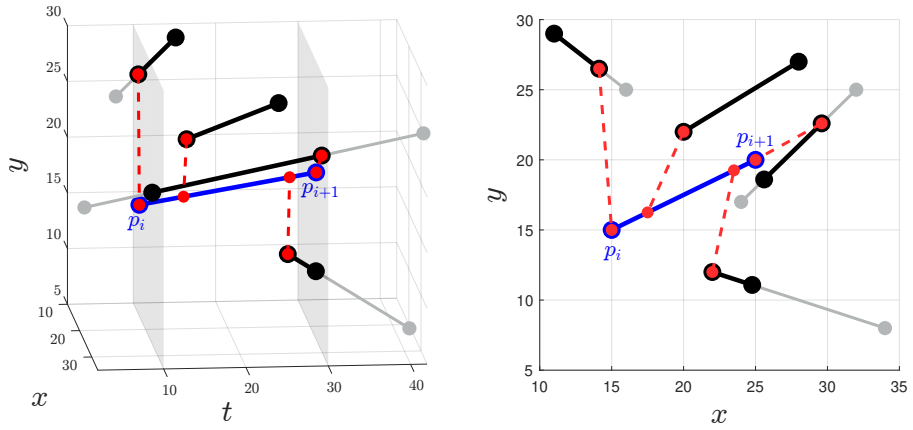


Figure 3.2: Example of minimum time matched distance (red). Segments in Q (black) are matched with segment $\overline{p_i p_{i+1}}$ (blue) between $[t_i, t_{i+1}] \in P$. Segments or parts of the segments that are outside of the duration are not considered (grey). *Left* figure shows the distances with the time dimension, and the *Right* figure shows the results in the spatial dimensions only.

For the time match exact distance join, we need to modify the join predicate. The spatial distance between line segments is calculated at a common time instance shown in Figure 3.2. First, any segment or parts of a segment outside of $[t_i, t_{i+1}] \in P$ is not considered. Then the minimum distance from $\overline{p_i p_{i+1}}$ to each segment in Q is computed. The line segments are spatially and temporally close if the distance is less than ϵ. Note that the time match distance is not necessarily equal to the minimum spatial distance. Formally, the minimum time-matched distance is described as:

$$minDistTime(\overline{p_i p_{i+1}}, \overline{q_j q_{j+1}}) = \min\{|p_t, q_t|\} \text{ where } p_t \in \overline{p_i p_{i+1}} \text{ and } q_t \in \overline{q_j q_{j+1}} \text{ and } \max(t_i \in P, t_j \in Q) \leq t \leq \min(t_{i+1} \in P, t_{j+1} \in Q) \quad (3.4)$$

Where, p_t is a point at time instance t on line segment $\overline{p_i p_{i+1}}$, q_t is a point at time instance t on line segment $\overline{q_j q_{j+1}}$, and $|p_t, q_t|$ is the distance between points p_t and q_t. We denote p_{tc} as the point on $\overline{p_i p_{i+1}}$ closest to point q_{tc} on $\overline{q_j q_{j+1}}$ and tc as the time instance where the minimum time matched spatial distance occurs.

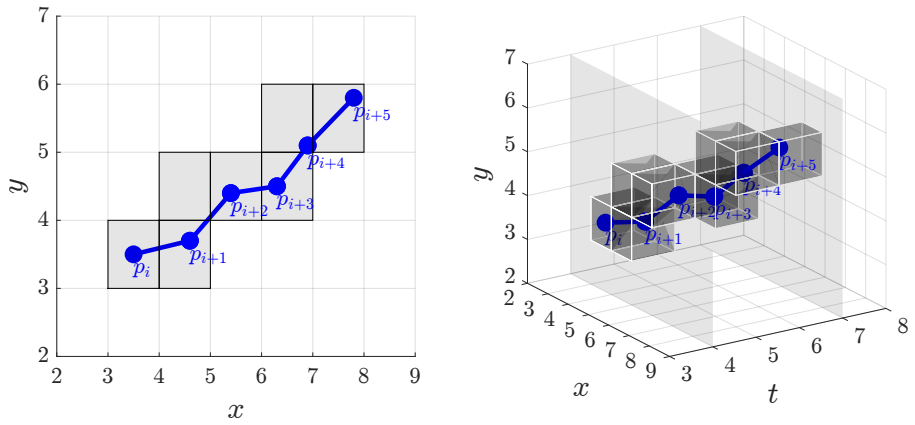


Figure 3.3: Example showing the difference in the number of grid intersections between segments and the line segment. A segment can spatially intersect 7 grid cells and 3 time slices, while a line segment intersects at most 3 grid cells and 2 time slices.

3.4 Trajectory Distance Join Algorithm

3.4.1 Design Overview

In this section, we present our framework for efficiently answering distance joins based on line segments. Our framework uses a filter-and-refine approach to execute the distance join operation. The filter-and-refine approach uses relatively low computation costs to prune out results that are impossible to return before executing more expensive distance computations. This approach is adopted by many studies mentioned in Section 3.2 [205, 206, 181, 12, 95, 195, 216, 11].

We use a uniform grid to partition the line segments. The two key insights are that (1) the spatial and temporal components are not correlated, and (2) line segments generate fewer duplicates. A simple example of how the spatial and temporal components are not necessarily correlated (as stated in Chapter 1) is a person walking within a city block. We need a higher temporal resolution compared to the spatial resolution since the spatial domain is confined. Another example of the opposite case is a short-distance flight, where we need a higher spatial resolution than the temporal resolution. Therefore, we treat the spatial and temporal grid resolution separately.

The second insight is that line segments generate fewer intersections with grid cells than segments (used in other studies in Section 3.2). We show an example in Figure 3.3 where a segment takes up 7 spatial grids and 3 time slice and duplicates to 9 partitions. The longest line segment only takes up 3 spatial grid and 2 time slice and requires only 3 duplicates. When storing minimum bounding boxes (MBBs), the difference in the number of duplicates will further increase (45 duplicates for the segment vs 8 for the line segment). Furthermore, storing and computing the distance between segments is much more expensive than line segments, as results may need to be aggregated between partitions. Therefore, we store trajectories as a sequence of line segments for efficient distance joins.

We further improve the join algorithm's performance at each stage of the filter-and-refine approach. During the filtering stage, we correlate the grid size and distance threshold to reduce duplicates. We prevent updating the grid size for new distance thresholds by considering historical distance thresholds and time windows. During refining, we build a local grid index that adjusts itself to output true positive results without distance computations. We now present each phase in detail.

3.4.2 The Partitioning Phase

The first step before any querying is to decide how the data is split and distributed across the nodes. Our design is based on uniform grid partitioning that partitions the space into equal-sized cells. It is fast to build and allows us to identify intersecting cells efficiently. We first introduce the concepts of uniform partitioning, starting with an example using the spatial domain. Specifically, the cell width of a grid cell in the x-dimension is the following:

$$w_x = \frac{x_{\max} - x_{\min}}{n_s} \tag{3.5}$$

Where, w_x is the cell width for the x-dimension, x_{max} and x_{min} are the maximum and minimum x value for the entire region. n_s is the grid resolution. Note that n is constant regardless of the (spatial) dimension. Each trajectory point can be assigned to a grid cell:

$$g_x = \frac{x - x_{\min}}{w_x} \tag{3.6}$$

Where, g_x is the grid index in the x-dimension. We form the 1D key by linearly combining the grid index in each dimension.

$$g_s = g_x + g_y \times n_s + g_z \times n_s^2 \tag{3.7}$$

g_s is the spatial grid index for a grid cell, g_x, g_y and g_z are the grid index in the x, y and z dimension respectively. Here, we use n_s to identify each dimension, similar to a hash function. For example, if n_s is 10, the ones-digit represents g_x, the tens-digit represents g_y, and the hundreds-digit represents g_z. A similar approach for temporal partitioning is known as time slicing. The temporal dimension is divided uniformly to form equal size time intervals:

$$w_t = \frac{t_{\max} - t_{\min}}{n_t} \tag{3.8}$$

Where, w_t is the time interval, $t_{\max}$ and $t_{\min}$ are the maximum and minimum time instances. n_t is the temporal resolution. The temporal grid index for a trajectory point can be calculated by:

$$g_t = \frac{t - t_{\min}}{w_t} \tag{3.9}$$

We combine the temporal and spatial dimensions to create a spatiotemporal index.

$$g_{st} = g_x + g_y \times n + g_z \times n_s^2 + g_t \times n_s^3 \tag{3.10}$$

g_{st} is the spatiotemporal grid index. It is used as the key for partitioning. We show how a single trajectory is partitioned in Algorithm 1. After splitting trajectories into line segments,

Algorithm 1 Grid Partitioning

```
1: Variables
2:     G_st ← {g_st, {p_i p_{i+1}}}                          ▷ Collection of Partitioned Line Segments
3: end Variables

4: procedure PARITIONTRAJECTORY(T_l)
5:     Input T_l                                             ▷ T_l - Trajectory from collection of line segments
6:     for p_i p_{i+1} in T do
7:         assignPartition(p_i p_{i+1})
8:     end for
9: end procedure

10: procedure ASSIGNPARTITON(p_i p_{i+1})
11:     Input p_i p_{i+1}                                    ▷ p_i p_{i+1} - Line Segment
12:     x_lower, x_upper ← min(x_i, x_{i+1}), max(x_i, x_{i+1})
13:     y_lower, y_upper ← min(y_i, y_{i+1}), max(y_i, y_{i+1})
14:     t_lower, t_upper ← min(t_i, t_{i+1}), max(t_i, t_{i+1})
15:     g_st,lower ← findPartition(x_lower,y_lower,t_lower)
16:     g_st,upper ← findPartition(x_upper,y_upper,t_upper)
17:     g_st,current ← g_st,lower
18:     while g_st,current ≤ g_st,upper do
19:         G_st[g_st,current].insert(p_i p_{i+1})
20:         g_st,current+ = 1
21:     end while
22: end procedure

23: procedure FINDPARTITION(x_i,y_i,t_i)
24:     Input x_i,y_i,t_i                                    ▷ x_i - Longitude, y_i - Latitude, t_i - Time instant
25:     g_x ← (x_i − x_min)/w_x
26:     g_y ← (y_i − y_min)/w_y
27:     g_t ← (t_i − t_min)/w_t
28:     return g_x + g_y × n_s + g_t × n_s^2
29: end procedure
```

we iterate over all line segments and assign partitions to each line segment in lines 6-7. Each line segment is partitioned by the *assignPartition*() function. In the *assignPartition*() function, the points are partitioned with the *findPartition*() function, which applies the concepts introduced above. Since the line segment may intersect multiple grid cells, we must assign the line segment to each grid cell (lines 19-20). The number of intersects and duplicates depends on the grid's resolution and the line segment's length. We use an MBB approximation instead of finding exact grid intersections to reduce the cost of partitioning.

3.4.3 The Filtering Phase

Once the trajectories are partitioned, the line segments in Q that are not close to P are filtered and will not participate in the later stages. E.g., segments that do not belong to the same spatiotemporal grid index. However, since we are performing a distance join, grid indexes close to the line segments must also be considered. We use approximate grid cell distances instead of actual distances to reduce the number of grid intersects and introduce the concept of rings.

Rings around a grid cell represent the level of adjacency of the neighboring grid cell. For example, if a grid cell has two rings, the inner ring contains first-degree neighboring cells, and the outer ring contains second-degree neighbors. We can add a single ring of grid cells around the original line segment to achieve the lowest number of duplicates. However, the minimum cell width must equal the distance threshold to prevent false negatives. Therefore, the grid resolution can be computed by rearranging Equation 3.5:

$$n_s = \min\left(\frac{i_{\max} - i_{\min}}{w_i}\right) \text{where, } i \in \{x, y\} \tag{3.11}$$

Note that adding a single ring may cause the grids to be susceptible to skewness issues. For example, if we end up with cells with larger sizes, the number of duplicates lowers, but each partition may be overloaded with a dense number of line segments. On the other hand, a smaller grid cell size generally decreases partition size but will cause line segments to be duplicated in more partitions [195]. The optimal grid size is data-specific and depends on the distribution of the spatial points and the maximum and minimum spatial values. Therefore, we extend Equation 3.11 to allow the grid resolution to be defined as factors of the distance threshold:

$$n_s = \min\left(\frac{i_{\max} - i_{\min}}{w_i \times factor}\right) \text{where, } i \in \{x, y\} \tag{3.12}$$

Now, the grid resolution can be changed depending on the data distribution. If the factor is greater than 1, the minimum width in each cell is larger than the distance threshold ϵ. Therefore, adding a single ring is sufficient. If the factor is less than 1, the minimum width is less than the ϵ, and outer rings are needed. For example, if the factor is 0.5, two rings need to be added, as the minimum width in each is only half of ϵ. A smaller factor will increase the number of duplicates, but it may benefit the system's overall performance.

For the temporal dimension, we find all temporal slices that intersect with the time window. We do not consider rings for the temporal dimension, which tends to grow continuously and may differ for the three distance join variants (shown in Section 3.5.2). To reduce the number of

Algorithm 2 Filtering

```
1: procedure FILTERTRAJECTORIES(P, Q)
2:     Input P, Q                                   ▷ P, Q - Dataset of Trajectories
3:     setPartition(ϵ, factor)
4:     for P_i in P do
5:         P_li ← assignLineSegment(P_i)
6:         G_st,p ← partitionTrajectoryExtend(P_li)
7:     end for
8:     for Q_i in Q do
9:         Q_li ← assignLineSegment(Q_i)
10:        G_st,q ← partitionTrajectory(Q_i)
11:    end for
12:    groupedSegments ← G_st,p.intersectGroupFilter(G_st,q)
13:    return groupedSegments
14: end procedure

15: procedure SETPARTITION(ϵ, factor)
16:    Input ϵ, factor
                                   ▷ ϵ - Distance threshold, factor - Multiples of ϵ
17:    x_part ← (x_max − x_min)/(ϵ · factor)
18:    y_part ← (y_max − y_min)/(ϵ · factor)
19:    n_s ← min(x_part, y_part)
20:    set w_x = (x_max − x_min)/n_s
21:    set w_y= (y_max − y_min)/n_s
22: end procedure

23: procedure ASSIGNLINESEGMENT(T)
24:    Input T                                      ▷ T - Trajectory
25:    p_i p_i+1 ← { }
26:    for i in [1 to length(T) −1] do
27:        p_i p_i+1.insert(lineSegment(p_i, p_i+1))
28:    end for
29:    return p_i p_i+1
30: end procedure

31: procedure PARTITIONTRAJECTORYEXTEND(T_l)
32:    Input T_l                                    ▷ T_l - Trajectory from collection of line segments
33:    for p_i p_i+1 in T do
34:        assignPartitionExtended(p_i p_i+1)
35:    end for
36: end procedure
```

potential duplicates, we use the average sampling duration of the dataset as our initial estimate for the time interval w_t. The average sampling duration is calculated by taking the average time duration between consecutive points and is determined beforehand.

As new queries come in, the distance threshold ϵ will change. We express the new ϵ in terms of the current grid cell distance. To avoid constant re-partitioning, the grid resolution will be updated only if the number of cell intersections is large (e.g., 2× more intersections than the previous join). The updated grid resolution is determined by averaging the historical distance thresholds to ensure good performance under different query loads. Although we do not use rings for the temporal dimension, we still update the temporal resolution based on historical query distribution with a similar strategy to the spatial update strategy. This method ensures we get the minimum number of duplicates when partitioning.

Algorithm 3 In Partition Grid Index

```
1: procedure ASSIGNINDEX(p_i p_{i+1})
2:     Input p_i p_{i+1}                                   ▷ p_i p_{i+1} - Line Segment
3:     x_lower, x_upper ← min(x_i, x_{i+1}), max(x_i, x_{i+1})
4:     y_lower, y_upper ← min(y_i, y_{i+1}), max(y_i, y_{i+1})
5:     t_lower, t_upper ← min(t_i, t_{i+1}), max(t_i, t_{i+1})
6:     g_{st,lower} ← findPartition(x_lower,y_lower,t_lower)
7:     g_{st,upper} ← findPartition(x_upper,y_upper,t_upper)
8:     if g_{st,lower} == g_{st,upper} then
9:         G_st ← g_{st,lower}
10:    else if only g_{x,lower} != g_{x,upper} then
11:        G_st ← horizontalLineIntersect(g_{st,lower}, g_{st,upper})
12:    else if only g_{y,lower} != g_{y,upper} then
13:        G_st ← verticalLineIntersect(g_{st,lower}, g_{st,upper})
14:    else if only g_{t,lower} != g_{t,upper} then
15:        G_st ← timeExtrudeIntersect(g_{st,lower}, g_{st,upper})
16:    else if only one of the dimensions equals then
17:        G_st ← rayTracingIntersect2D(g_{st,lower}, g_{st,upper})
18:    else
19:        G_st ← rayTracingIntersect3D(g_{st,lower}, g_{st,upper})
20:    end if
21:    return G_st
22: end procedure
```

Algorithm 2 shows the entire filtering stage. We set the grid resolution in line 3 to ensure the width of the grid cell is always greater than or equal to ϵ. We then convert data points into line segments and partition each line segment. One of our datasets (e.g., P) is extended to include the neighboring cells in line 6. Next, line segments with common grid indexes are grouped and placed into partitions. Grid cells not intersected by line segments in P and Q are filtered.

3.4.4 The Refining Phase

The refining stage filters out false positives and finds the trajectory pairs with distances less than ϵ. We use a reference point de-duplication scheme [61] to avoid cross-node communication and post-join pass-through for removing replicated line segments. Any local join algorithm can be used in each partition, where we adopt the commonly used sweep-line algorithm as the local join algorithm.

We improve the refining process by reducing the number of distance computations. The idea is to output true positives directly with a local on-the-fly in-memory grid index. The local index is built in each partition and will change depending on ϵ. The local grid resolution $n_{l,s}$ is computed to ensure the maximum distance within a grid cell is equal to the distance threshold. Since the maximum distance in a rectangular grid cell is the diagonal distance, we can calculate

$n_{l,s}$ by making the diagonal distance equal to ϵ. However, this means the index may need to be rebuilt every time the query radius changes. To reduce the number of rebuilds, we propose to index based on the minimum historical distance thresholds. This ensures any matched results are spatially close. Even if rebuilding is needed, the grid index is one of the fastest among indexing structures (especially compared to hierarchical structures). The local time resolution $n_{l,t}$ is computed by taking the time duration ($w_{l,t}$) in each grid cell to equal d. Similarly, the minimum time window is used. These two conditions ensure the line segments in the same grid index are true-positives.

The exact intersections are needed instead of the MBB approximation to find true-positive results. We propose a new indexing technique based on ray-tracing [16] shown in Algorithm 3. The *assignIndex*() function is used to find the exact intersection between a line segment and grid cells (Figure 3.3).

Algorithm 4 Time window distance joins

```
1: procedure REFINEWINDOW({p_i p_{i+1}}, {q_j q_{j+1}}, d, exact)
2:     BuildGridIndex(p_i p_{i+1}, q_j q_{j+1})
3:     setIndex(ε, d)
4:     for p_i p_{i+1} ∈ {p_i p_{i+1}} do
5:         G_{st,p} ← assignIndex(p_i p_{i+1})
6:         if exact then
7:             if G_{st,p}.intersects(G_{st,q}) then
8:                 Output Pair(p_i p_{i+1}, q_j q_{j+1})
9:             else if minDistExact(p_i p_{i+1}, q_j q_{j+1}, ε, d) then
10:                Output Pair(p_i p_{i+1}, q_j q_{j+1})
11:            else
12:                Emit Pair(p_i p_{i+1}, q_j q_{j+1})
13:            end if
14:        else
15:            for q_j q_{j+1} ∈ tempIntersect(p_i p_{i+1}, {q_j q_{j+1}}, d)
   do
16:                G_{st,q} ← assignIndex(q_j q_{j+1})
17:                if G_{st,p}.intersects(G_{st,q}) then
18:                    Output Pair(p_i p_{i+1}, q_j q_{j+1})
19:                else
20:                    if minDist(p_i p_{i+1}, q_j q_{j+1}) ≤ ε then
21:                        Output Pair(p_i p_{i+1}, q_j q_{j+1})
22:                    else
23:                        Emit Pair(p_i p_{i+1}, q_j q_{j+1})
24:                    end if
25:                end if
26:            end for
27:        end if
28:    end for
29: end procedure

30: procedure SETINDEX(ε, d)
31:     n_{l,s} ← sqrt(((x_max − x_min)^2 + ((y_max − y_min)^2))/ε^2)
32:     set w_{l,x} = (x_max − x_min)/n_{l,s}
33:     set w_{l,y} = (y_max − y_min)/n_{l,s}
34:     set w_{l,t} = d
35: end procedure

36: procedure TEMPINTERSECT(p_i p_{i+1}, {q_j q_{j+1}}, d)
37:     for q_j q_{j+1} ∈ {q_j q_{j+1}} do
38:         if !([t_i − d, t_{i+1} + d] ∩ [t_j, t_{j+1}]) where i ∈ P and
   j ∈ Q then
39:             emit q_j q_{j+1} from {q_j q_{j+1}}
40:         end if
41:     end for
42:     return {q_j q_{j+1}}
43: end procedure

44: procedure MINDISTEXACT(p_i p_{i+1}, q_j q_{j+1}, ε, d)
45:     P_c, Q_c ← minDistance3D(p_i p_{i+1}, q_j q_{j+1})
46:     if spatialDist(P_c, Q_c) ≤ ε & temporalDiff(P_c, Q_c) ≤ d
   then
47:         return true
48:     end if
49: end procedure
```

3.4.5 Time Window Distance Join and Exact Distance Join

We present the time window distance join process with Algorithm 4. The distance join is applied to each partition and is processed in parallel. The local grid index is generated lazily. Line 3 ensures the maximum distance in each grid cell is less than or equal to the distance threshold ϵ and the time duration in each grid cell is less than or equal to the time window d. Then, all line segments that temporally intersect are found, and the rest are filtered. Line segments with the same grid index are outputted while the rest undergoes distance computations.

For time window exact distance join, instead of computing temporal and spatial distance computations, we find the closest points between the pairs. It is considered a match if the closest points meet the spatial and temporal criteria.

Algorithm 5 Time match distance join

```
1: procedure refineTimeMatch({p_i p_{i+1}}, {q_j q_{j+1}}, d)        ▷ {p_i p_{i+1}},{q_j q_{j+1}} - Line segments ∈ P and Q respectively
2:     for p_i p_{i+1} ∈ {p_i p_{i+1}} do
3:         for q_j q_{j+1} ∈ temporalIntersect(p_i p_{i+1}, {q_j q_{j+1}}) do
4:             if minDistTimeMatch(p_i p_{i+1}, q_j q_{j+1}, ϵ) == true then
5:                 Output Pair(p_i p_{i+1}, q_j q_{j+1})
6:             else
7:                 Emit Pair(p_i p_{i+1}, q_j q_{j+1})
8:             end if
9:         end for
10:    end for
11: end procedure

12: procedure temporalIntersect(p_i p_{i+1}, {q_j q_{j+1}})
13:    for q_j q_{j+1} ∈ {q_j q_{j+1}} do
14:        if !([t_i, t_{i+1}] ∩ [t_j, t_{j+1}]) where i ∈ P and j ∈ Q then
15:            emit q_j q_{j+1} from {q_j q_{j+1}}
16:        end if
17:    end for
18:    return {q_j q_{j+1}}
19: end procedure

20: procedure minDistTimeMatch(p_i p_{i+1}, q_j q_{j+1})
21:    p_{t_s} p_{t_e}, q_{t_s} q_{t_e} ← commonTimeInstances(p_i p_{i+1}, q_j q_{j+1})
22:    minDist ← spatialDistance(p_{t_s}, q_{t_s})
23:    for t ∈ (t_s, t_e] do
24:        P_t ← interpolate(p_{t_s} p_{t_e}, t)
25:        Q_t ← interpolate(q_{t_s} q_{t_e}, t)
26:        if minDist < spatialDistance(p_t, q_t) then
27:            minDist ← spatialDistance(p_t, q_t)
28:        end if
29:    end for
30:    if minDist ≤ ϵ then
31:        return true
32:    end if
33: end procedure
```

3.4.6 Time Match Exact Distance Join

The time match distance join process is in Algorithm 5. We don't need the local grid index since this method has a strict temporal condition. We first apply a temporal filter and then a spatial filter. Note that the duration of each line segment is not expanded for this method. The common time interval between two line segments is found (Figure 3.2). The pair of line segments is outputted if the minimum distance at common timestamps is less than ϵ.

3.5 Experimental Evaluation

In this section, we evaluate our framework against the existing method (GeoAnalytics [195] bin join method), which stores the trajectories as a sequence of points. We first show the performance for various system parameters (spatial and temporal resolutions). We then show how our framework performs under varying query parameters before showcasing its scalability.

3.5.1 Experimental Setup

The trajectory dataset is distributed on a Spark cluster. The cluster consists of 4 nodes, each equipped with a dual-core 1.5GHz CPU, 16GB of main memory (11GB of the main memory is available to Spark), and 25GB of secondary storage running Ubuntu 14.04.

The trajectory dataset used in the experiments is the Danish vessel trajectory [20] dataset, which contains over 1.8TB of automatic identification system (AIS) messages over Danish waters since 2006. The dataset's storage overhead far exceeds our cluster's memory capability. Only parts of the data are thus used for the experiments. The experiments were conducted on a 10-day span of AIS data, accounting for around 14000 trajectories with 48 million points and a size of 3.57 GB. The trajectory distance join is implemented on Spark using Scala. The experiments were carried out as self-joins.

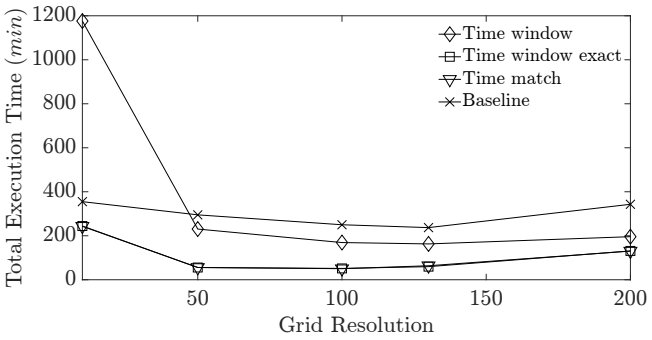


Figure 3.4: Performance of distance join with varying grid resolution.

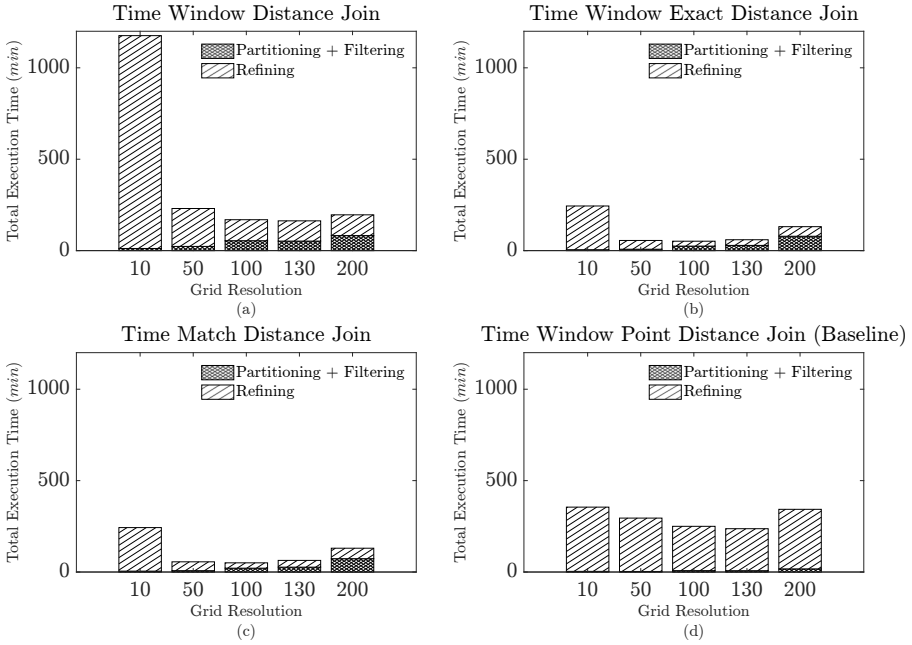


Figure 3.5: Run time breakdown of distance join with varying grid resolution.

From the dataset, an initial average duration of 300s was found and is used as the default time window d, which results in 508 temporal slices. The distance threshold ϵ of 5km is used, resulting in 130 grids per dimension. These parameters are varied in Section 3.5.3. We used the Haversine distance for AIS longitude and latitude values.

3.5.2 System Parameters Evaluations

The spatial and temporal resolutions are very crucial parameters. Temporal resolution is especially important as it controls the number of spatial partitions accessed. Having the optimal time slices reduces the number of spatial partitions accessed.

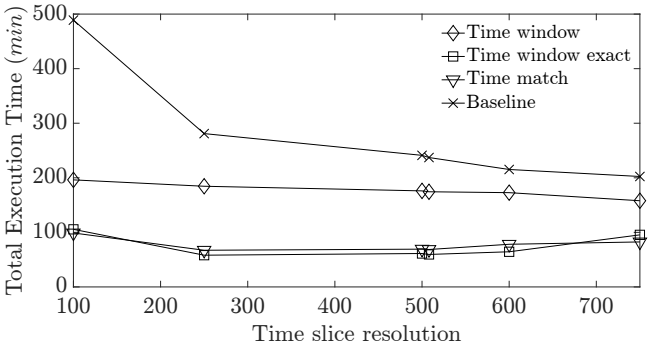


Figure 3.6: Performance of distance joins with varying number of time slices

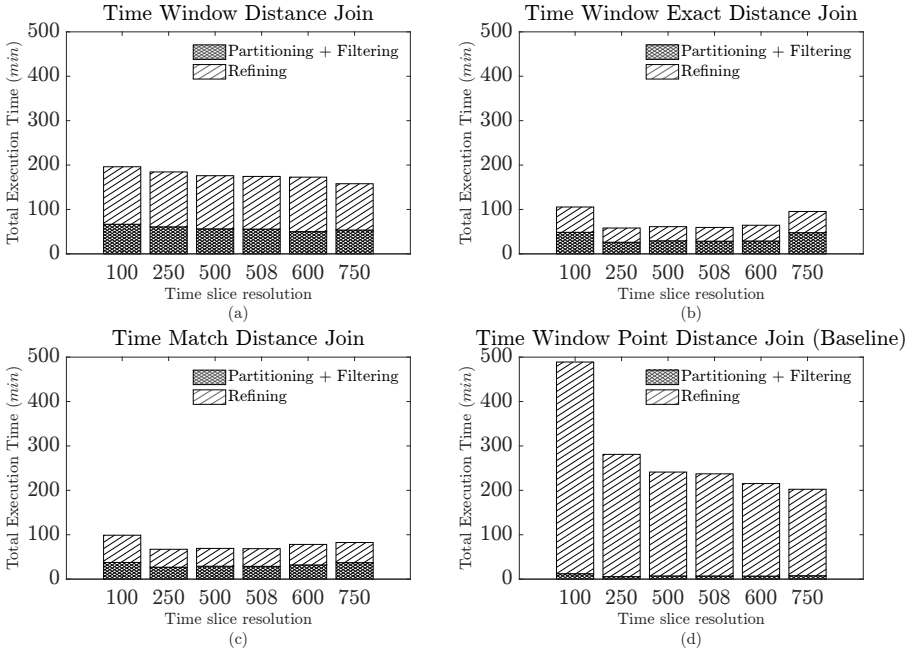


Figure 3.7: Run time breakdown.

Effect of Spatial Grid Resolution: The experiment was conducted by fixing the number of temporal slices and varying the grid resolution. Figure 3.4 shows the execution time for each method, and Figure 3.5 shows the execution time for each processing stage. The results show that increasing the grid resolution is beneficial for the performance of the join. However, increasing the grid resolution too high deteriorates our performance. The results coincide with typical grid behavior, where large grid cells result in a smaller number of partitions but suffer from some partitions being overloaded. The majority of the time is spent pruning out false positives in the refining stage (Figure 3.5). On the other extreme, very fine grid cells lead to excessive replication, which requires more partitions to be explored to answer a query. Hence increasing the portion of the time spent in the filtering stage (Figure 3.5). The ideal grid resolution is close to our default value ($\epsilon = 5km, d = 300s$), suggesting that our grid resolution

estimation method (Equation 3.11) gives an accurate approximation of the ideal grid resolution given ϵ.

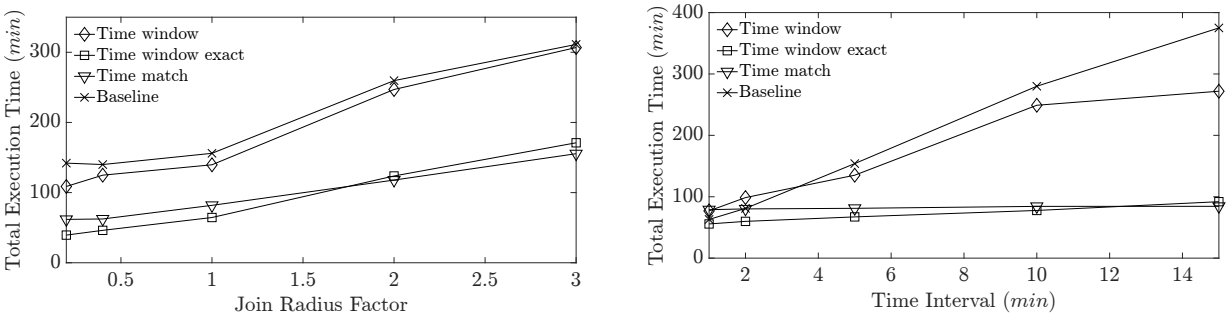
In addition, we compare the efficiency of our proposed methods and the baseline point method. We observed that all of our methods outperform the baseline point method in the optimal range (Grid Resolution $= 50 - 150$). We attribute this to using line segments over points. Line segments have far fewer duplicates in different partitions and have smaller partition sizes. The filtering stage for the point method, on average, takes 2.6% of the total execution time shown in Figure 3.5d. The time window and baseline methods are directly comparable, highlighting the efficiency of point distance computations compared to line segment distance computations when comparing the grid resolution at 10 (both methods fail to filter out false negatives). The time window exact method and the time match method are considerably more efficient due to even smaller partitions. Both methods have a stricter join criterion, which requires temporal intersections between the line segments. Hence, the candidate set is much smaller than the baseline method.

Effect of Temporal Grid Resolution: The experiment was conducted by fixing the spatial resolution and varying the temporal resolution. Figure 3.6 shows the execution time for each method, and Figure 3.7 shows the execution time for each processing stage. Similar to the grid resolution, very coarse time resolutions result in poor performance. The proportion of time spent in the filtering stage is around the same among all the methods. This suggests the cost of the filtering stage is mostly dependent on the granularity of the spatial resolution and that there are fewer duplicates in the temporal dimension. Figure 3.6 shows the overall temporal resolution range (x-axis) is much higher than the spatial grid resolution range. Although this observation may depend on the trajectory dataset's temporal density, it does suggest that having more temporal slices is beneficial for performance. These observations show that temporal updates are generally more uniform and less susceptible to hot spots (skewness) compared to the spatial dimension.

The optimal temporal resolution differs for each method, unlike the optimal grid resolution. Figure 3.6 shows that the line segment methods consistently outperform the baseline point

Table 3.1: Optimal configuration for each distance join method.

Methods	Data Structure	grid size n_s	time slice n_t
1. Time window	Line Segment	130	750
2. Time window exact	Line Segment	100	508
3. Time match	Line Segment	100	508
4. Baseline	Point	130	750



(a) Performance of distance join with varying distance threshold sizes as a factor of the default distance threshold.

(b) Performance of distance join with varying time window sizes.

Figure 3.8: Query and Data Parameters Evaluations

method. Similar to the grid resolution results, we attribute this to having fewer segments compared to the baseline. Interestingly, the baseline benefits significantly from increasing the temporal resolution. We believe this is caused by the skewness in the spatial data distribution, where a spatially dense area results in a large number of points being assigned to one partition. Higher temporal resolution helps alleviate this problem by reducing the size of each partition, as dense areas are often distributed more evenly in time. Line segments do not seem to benefit substantially from having an optimized temporal resolution. Instead, it benefits more from having an optimized spatial resolution. On the contrary, the point method does not gain substantial performance from an optimized spatial grid; an optimized temporal grid is much more effective at reducing skewness and overloaded partitions.

3.5.3 Query and Data Parameters Evaluation

Table 3.1 shows the optimal configuration for the rest of the sections.

Effect of Distance Threshold: The effect of the distance threshold ϵ on the performance of the trajectory distance join is in Figure 3.8a. Naturally, when ϵ increases, the running time of each algorithm increases as it needs to spend more time querying more trajectories. The

line segment methods perform better than the baseline across all ranges. When comparing the time window distance join and baseline, we see that the gain in performance reduces when increasing ϵ. The decrease in performance is because the line segment's minimum bounding rectangle (MBB) grows much faster than a point's MBB—a larger MBB results in dead space. Line segments located in dead spaces cannot be pruned out by the local grid index in the refining stage, as the two line segments do not intersect (spatially and temporally). Hence, more dead space means more line segment comparisons. Furthermore, as more points need to be compared, a point-to-point distance computation is far less expensive compared to a line-to-line comparison. We see that the time window exact method performs better than the time match method for smaller ϵ. This gain in performance can also be attributed to the effectiveness of the local grid index in the refining stage, which the time match method does not have.

Effect of Time Window: Figure 3.8b shows the performance of joins with different time windows d. We can see a linear dependence on the execution time and the time window size for all methods except for the time match method. Similar to increasing the distance threshold, increasing d introduces more points/line segments, thereby increasing the execution time. The time match method is expected to stay constant since it does not depend on the time window. The point method performs extremely well for small time intervals due to having less overloaded partitions. The line segment methods perform better over the rest of the range. Some performance gains can be attributed to using the index grid index in the refining stage. However, most of the difference comes from the high inefficiency of the baseline at larger time windows.

3.5.4 Scalability Evaluation

We evaluate the scalability of our framework through two tests. The first test increases the dataset size (scale up), and the second test increases the number of nodes (speed up) in our cluster while keeping the dataset size the same.

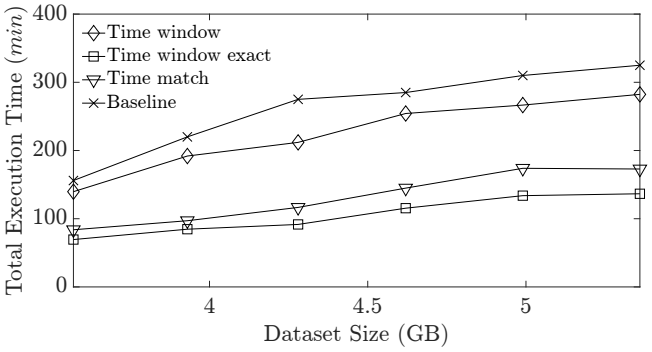


Figure 3.9: Performance of distance join with increasing dataset size.

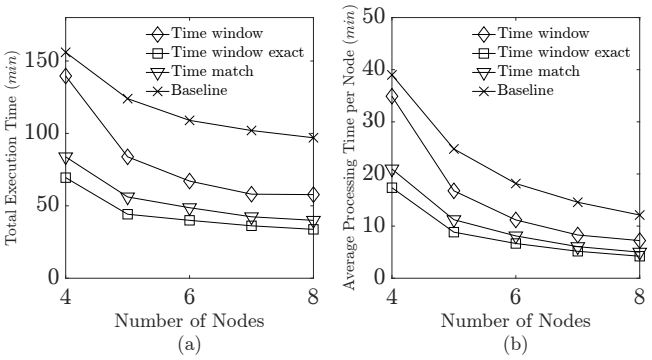


Figure 3.10: Performance with increasing number of nodes. (a) Total Time. (b) Average Execution Time per Node.

Effect of Dataset Size: Naturally, the results in Figure 3.9 show an increase in execution time as the dataset grows. The slope of the time window exact and time match methods are relatively flat, which shows good scalability for increasing data size. The difference in execution time between the line segments and the baseline method is about equal across all dataset sizes, suggesting that using line segments, along with the local grid index, can achieve similar performance gains over the baseline method regardless of data size.

Effect of number of nodes: One very important aspect of an efficient distributed system is its' parallel processing ability. Figure 3.10a shows the total execution time against various computing nodes, while Figure 3.10b shows the average execution time per node. The results show a decrease in execution time, as expected. Increasing the number of nodes increases the number of partitions that can be processed in parallel, which reduces the execution time. As the number of nodes increases, the gain in performance decreases among all the algorithms due to increasing shuffling costs between distributed nodes. One interesting observation is that the point method seems to have a slightly slower rate of change (less performance gain over nodes).

As we concluded in the previous sections, the cost of the filtering stage for the point method is negligible compared to the refining stage, and the cost of line-to-line comparisons is more expensive than point-to-point comparisons. Therefore, this means that each point partition must have significantly more points than their line segment counterparts. Hence, using line segments is much more efficient than points for trajectory distance join operations.

3.6 Summary

In this chapter, we study the distance join for trajectory data and present three methods of answering distance join queries in a distributed environment. We propose to use line segments instead of points to represent trajectories. Additionally, we introduce an in-partition lazy grid index to more efficiently find true positives and reduce the number of distance computations in each partition. Extensive experiments show that our line segment methods outperform the point method baseline, and the in-memory grid index scales well with increasing data size and computing nodes. Specifically, we are able to improve total processing time by up to 43× compared to the state-of-the-art point-based method.

Chapter 4

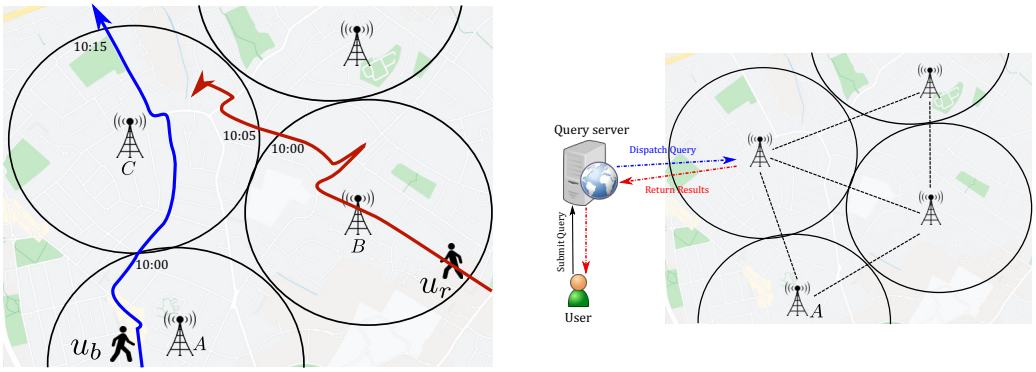
In-Network Privacy-aware Range Queries

4.1 Overview

In Chapter 4, we introduce our framework for tackling the challenge of privacy-aware trajectory queries from a global perspective. These queries are becoming more crucial with the increase in user awareness of data privacy and data protection legislation. We specifically study the problem of the privacy-aware range query, ubiquitous across location-based services [70, 182, 185, 82, 77]. Instead of returning a subset of objects, a privacy-aware range query aggregates the total number of distinct moving objects in a spatial region over a time interval. The main challenge for privacy-aware queries is tracking trajectories during updates since the object identifiers are not stored.

Existing systems [41] preserve privacy by continuously aggregating data at a central server before answering queries. This allows them to get rid of the identifiers after aggregating and benefit from the extensive mature centralized algorithms [70, 182, 33, 82] since all the data is in the same place*. However, centralized systems cannot efficiently deal with high rates

*More rigorously, the same place means the systems can easily access the full aggregate history of the data. It does not limit the system to place all data in the same storage medium.



(a) (b)

Figure 4.1: (a) A sensors network tracking two users u_b and u_r. (b) The associated communication infrastructure.

of updates for real-time applications and often face communication bottlenecks. Extensive continuous communication is inevitable since updates must be sent to the central server for aggregation. Furthermore, centralized systems have a privacy concern since all data is stored in the same location before aggregation, making them more vulnerable to malicious attacks.

Recently, in-network systems have become increasingly popular as they mitigate the problems of centralized systems by naturally maintaining data locality. Updates are aggregated on relevant (close-by) sensors, and the central server only needs to communicate with sensors involved in the spatial query range. Figure 4.1 shows an example of a load-balancing application tracking the number of users in a cell tower network. Figure 4.1a shows the movement of two users and the sensing regions A, B and C. A spatiotemporal aggregate range query for sensing region C between 10:05 to 10:10 will return a count of 2. Privacy is preserved as all updates within C are local without needing the moving object's full history (e.g., movement in A or B). The nature of in-networks systems means no single party can access the full mobility pattern at any time. In terms of communication (shown in Figure 4.1b), queries are dispatched to the relevant sensors (only to C without needing to communicate to A or B). Communication with the centralized server (the query dispatcher) is needed only when querying, which is much more efficient than the continuous sync required for centralized systems.

However, in-network systems are not naturally adapted to handle moving object aggregation queries. It is especially challenging when objects are frequently updated as they suffer from double counting and dead space problems. The *double counting problem* occurs when we do

not store moving objects' identifiers, resulting in distinct objects being counted multiple times during updates [70, 182]. This is inevitable for privacy-aware queries as we cannot store the object identifiers. Compounding the double counting problem, the *dead space problem* generates excessive communication between sensors if the spatial region has ill-suited partitions. Unlike centralized systems, in-network systems are unaware of dead spaces in partitions. Therefore, they must flood all sensors within the region during querying, thereby increasing the communication cost (proportional to the area of the query region [157]).

We, therefore, develop a novel framework for efficient privacy-aware in-network aggregate queries. Our main design idea is to build a framework based on a non-axis-aligned spatial representation of the sensor network that can efficiently support moving object updates. We introduce the idea of storing both the incoming and outgoing counts at the edges of spatial partitions during updates, which solves the double-counting problem without unique identifiers. A planar graph representation configures the spatial partitions (based on sensor and edge locations) that consider both sensor and query distributions to minimize the dead spaces while preserving query accuracy.

Furthermore, to achieve efficient query processing (in terms of query efficiency and in-network sensor infrastructure cost), we propose to build the framework based on a subset of sensors. We show a host of sampling methods when the query distribution is unknown or dynamic and use a modified submodular maximization algorithm [114] when the query distribution is known. The sampled sensor configuration can also aid sensor deployment (e.g., to achieve the best cost-saving and query accuracy) or select a subset of sensors to activate at particular times to save energy for a pre-existing set of sensors.

We then present an associated query algorithm based on differential 1-form, which only requires accessing the sensors (or motes) on the perimeter of the query region to reduce communication. Specifically, counts are stored on the edges of the planar graph, and queries are answered by integrating the counts along the edges of the query region (defined as faces of the planar graph). To solve the double counting problem, we store two counts (incoming and outgoing) on each directed edge in the planar graph. When answering queries, the incoming and outgoing counts

will ensure objects that exit and enter disjoint partitions (or faces in the graph) multiple times will cancel out. To increase the query performance and reduce storage size on a single sensor, we take inspiration from learned indexing to infer the count at different timestamps using constant-sized machine-learning models. This paves the road for using learned indexes to index the temporal dimension of trajectories in Chapters 5&6.

4.2 Drawbacks of Existing Works

To the best of our knowledge, no existing methods bridge the gap between privacy-aware data aggregation and selecting a subset of sensors for efficient query processing. In the following section, we review existing centralized and in-network approaches for answering privacy-aware queries before presenting works on sensor selection.

4.2.1 Centralized Approaches

We first review the centralized approaches for answering privacy-preserving queries. To recap, range queries on spatiotemporal trajectories can be efficiently answered with hierarchical data structures introduced in Section 2.3.5. They can be extended support updates (without preserving privacy) on trajectories with lazy updates [4], incremental updates [35], or selective updates [191].

Data aggregation has been shown to be one of the most prominent methods for answering privacy-preserving queries. Some notable works on aggregating trajectories include distributed Euler-histograms [70, 82], tree structures [182], and holistic aggregate functions [33].

All of these methods require the full history of the moving objects prior to aggregation. As mentioned in Section 4.1, we argue that in-network systems can better preserve privacy compared to centralized approaches by keeping the data local to each sensor. Thus preventing any party from viewing the full history at any time.

4.2.2 In-Network Approaches

For in-network systems, the sensing region of each sensor can be abstracted to represent spatial subdivisions for querying. Traditionally, spatial subdivisions can be divided using axis-aligned spatial indexes, including, grids [70, 82, 122], R-trees [34, 182], k-d trees [5, 35], and quadtrees [199, 115]. However, these methods generate much dead space and result in a communication cost proportional to the area of the query region [80]. [157] improves the communication cost by using planar graphs and differential 1-forms to subdivide the space. They support spatial range queries but do not answer spatiotemporal queries. Our framework takes inspiration from [157]. While they focus on network performance, we extend this idea to efficiently answer queries with a subset of sensors and analyze the framework based on query performance.

4.2.3 Sensor Selection

Lastly, we review sensor selection techniques, which have been widely studied for sensor placement as a standalone problem. Some notable works include [24, 133], which optimize sensor placements for traffic flows, and [112], who proposed a submodular maximization algorithm for selecting sensors for contamination detection. The idea of sampling has also been used in the sensor placement realm, where [98] approximates a query by collecting samples in each sensor, and [38] reduces the number of sensors required to track movement between spatial subdivisions. However, as mentioned at the beginning of the section, no methods discuss how to track and aggregate movements for queries with a subset of sensors.

4.3 Background

In this section, we formally introduce the problem setting before clarifying the reasoning for our design. We then formulate the query and introduce technical details and mathematical terminologies used in the rest of the chapter.

4.3.1 Problem Setting

In-network systems store counts locally to the sensor responsible for the detection. Sensors can communicate with each other (e.g., mesh network), and some sensors act as communication nodes to manage and aggregate counts from nearby sensors [203].

They improve privacy and reduce communication under continuous updates over centralized systems. Privacy is guaranteed as data is kept local to relevant sensors, with no parties obtaining the full mobility pattern of moving objects at any time. As for communication, centralized systems continuously sync with all the sensors for up-to-date counts, where they need to manage a large number of simultaneous connections from each sensor. This may be superior when the frequency of queries is higher than updates. However, we argue that in most practical cases, the update frequency is much higher than the query frequency. In this case, in-networks systems excel with lower communication costs, which apply to all network scenarios (e.g., increase bandwidth from 6G). Furthermore, substantial network bandwidth and power are needed for centralized systems if sensors are far from the servers (e.g., high-power radios for long-distance data transmission, which can quickly drain battery-powered sensors).

However, we must address the dead space and distinct count problems to support privacy-aware aggregate count query processing for in-network systems. To create an efficient in-network query framework, we explore the idea of a planar graph with discrete differential forms and sensor placement techniques to decompose the spatial domain. Aside from solving these problems, we must also ensure efficient query performance (accuracy & time) and low sensor infrastructure cost (setup & operating cost). We now explain the two problems before justifying the reasoning and advantages of the design decisions.

Dead Space Problem: We first expand on the dead space problem described in Section 4.1. The *dead space problem* occurs when ill-suited spatial partitions generate unnecessary communication. The main rationale is that existing axis-aligned spatial partitioning techniques are designed for centralized systems. Therefore, the partition considers the spatial distribution of the entire data rather than the distribution of sensors.

For example, a sensor network is deployed for traffic monitoring in a city with curved roads (exemplary of real-world cities, except Manhattan). Suppose we apply axis-aligned partitioning based on the data distribution. In that case, we may place or select sensors (if a pre-existing set of sensors exists) at regular axis-aligned intervals (e.g., with an R-tree). This means we have sensors on roads and in areas between roads (where no vehicle is present). Processing queries would require the system to collect data from all the sensors within the spatial region, leading to unnecessary communication and computation. Additionally, this leads to a waste of resources as the sensors at the dead spaces consume power without any contribution.

Double Counting Problem: The *double counting problem* occurs when we cannot distinguish unique objects without identifiers and count an object multiple times when it exits and re-enters the sensing region during updates. Let's consider an example of a sensor network that monitors traffic flow on a highway. The sensors are placed at the exit and entry ramps. A vehicle will be counted every time it enters the highway and exits at the next ramp before re-entering it, resulting in overestimating the number of distinct vehicles.

4.3.2 Design Reasoning

We avoid ill-suited spatial partitions by introducing a planar graph representation of the sensor distribution. The nodes of the planar graph represent the sensors, while the faces represent the coverage of the sensors. The edges of the planar graph are communication paths between nodes and represent the border of the sensor coverage. The sensors at the nodes bordering an edge detect updates of moving objects when they move from one face to another. To prevent the double counting problem, we introduce two counts at the edges for each direction of travel (incoming & outgoing)[†]. To answer queries, we represent the query region as faces of the planar graph. The count is found by routing along the perimeter nodes of the faces. The planar graph allows us to route through the graph's edges to answer queries efficiently. An associated query algorithm is introduced to take into account the two counts when routing through the edges during aggregation.

[†]The exact techniques for finding the direction of moving objects in a sensor network can be found here [96].

To achieve efficient query processing, we need to consider the sensor configuration. We do not need every sensor to communicate with the remote query dispatching server. Instead, we chose a subset of communication sensors[‡] to reduce operating costs and communication costs (which also decrease query response time with less routing). The overall query efficiency is improved at a loss of query accuracy. The main challenge is how to minimize the communication sensors while preserving query accuracy.

We seek to tackle this challenge from the perspective of an in-network sensor placement problem. We propose a sensor sampling algorithm based on submodular maximization, which accounts for the query distribution to maximize the sensor coverage for a given number of sensors. We introduce a host of sampling methods to configure the sensors to support applications when the query distribution is unknown or dynamic.

4.3.3 Mobility & Sensing Domain

Before presenting the planar graph representations used in the query formulation, we first introduce the mobility and sensor domains for tracking moving objects.

Mobility Domain: The mobility domain is where moving objects are located and represents physical spatial regions, such as the city's roads. Vehicles move along the roads, and the layout of the road restricts movement. We usually represent this as a network graph. For example, roads represent edges, and junctions represent nodes.

Sensor Domain: The sensor domain represents the placement of sensors in a mobility domain used to track movements. In the road network example, we can place sensors at strategic points to track movement where the sensors' network connectivity graph represents the sensor domain.

[‡]The details of how communication sensors collaborate to track objects can be found here [202].

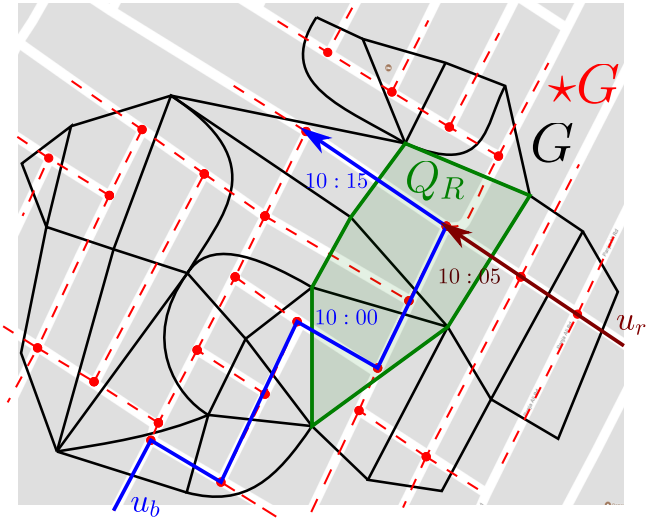


Figure 4.2: Example of road network: the mobility graph is $\star G$, dual graph G is the sensing graph. Two objects, u_b and u_r, are tracked within the region. Q_R is the query region (green). u_b enters the Q_R at 10:00 and leaves at 10:15. u_r enters the Q_R at 10:05 and does not leave.

4.3.4 Planar Graph Representation

Our framework abstracts the mobility and sensor domains as planar graphs for their properties and advantages. This is widely accepted as we can convert mobility graphs using spatial decomposition and overlapping sensing regions to non-overlapping "virtual" sensing regions (e.g., within the middleware).

The planar graph representation is shown in Figure 4.2. We denote the planar mobility graph as $\star G = (\star V, \star E)$. Using the road network example, $\star V$ are junctions, and $\star E$ are roads. The sensor graph is denoted $G = (V, E)$, where V are the sensors and E are communication links between the sensors. G is constructed to be the dual graph of $\star G$, which means the V represents a face in $\star G$. This allows us to use properties of the dual graph, such as vertex-edge duality.

We represent the movements using discrete differential 1-forms (which we will introduce in Section 4.3.6) on the edges of E. To answer a query, we convert the spatial range to be a union of faces in the sensor graph G, shown as Q_R in Figure 4.2. The count is found by routing the sensors around the perimeter of Q_R.

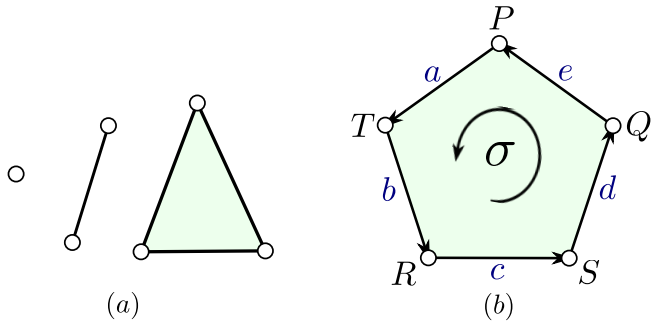


Figure 4.3: (a) 0,1,2 cells (b) An oriented face σ

4.3.5 Query Formulation

We now formulate the query with respect to the planar graph representation. Spatiotemporal range queries have both a spatial and time component. Given, the spatial region is Q_R, and the time interval $[t_1, t_2]$, we can formulate two types of queries:

Static object count query: Counts the objects that enter Q_R before time t_1 and leave after time t_2 (and does not temporarily leave Q_R). This counts the number of objects within the region during the time interval. E.g., in Figure 4.2, the count in Q_R between [10:05,10:10] is 2. This query strictly generalizes the spatial range count query, as studied in [157]. To answer the spatial case, we can set the timestamps t_1 and t_2 to be very close.

Transient object count query: Counts the objects that either enter Q_R before t_1 and leave during $[t_1, t_2]$; or enter Q_R between $[t_1, t_2]$ and leave after t_2. E.g., in Figure 4.2, the count in Q_R between [10:05,10:10] is one since u_b enters before 10:05 and leaves after 10:10. If we consider the time interval [10:00,10:03], u_r is not in the count. This query provides the net change of objects during the time interval, which is useful for applications such as traffic flow estimation [167], where the transient count is used to calculate the velocity of vehicles in a region (net count/time).

4.3.6 Discrete Differential Forms

Lastly, we introduce relevant concepts about cell complexes and discrete differential one-forms. In algebraic topology, points/vertices/nodes are considered 0-dimensional cells (0-cells),

lines/edges are 1-cells, and faces are 2-cells (example in Figure 4.3a). A cell complex is a collection of cells whose dimension is determined by the highest-dimension cell. E.g., the planar graphs $\star G$ & G in Figure 4.2 are considered 2-cell complexes.

The orientation of a k-cell is inherited from its $(k+1)$-cell. In Figure 4.3b, the face σ, orientated counter-clockwise, imposes the orientation of directed edge e to be $[Q, P]$. The same edge e in the opposite direction is $-e$, which will be $[P, Q]$ in our example. We orient the faces counter-clockwise for our convention [107].

A k dimensional chain C is a linear combination of k-cells c_i: $C = \sum \lambda_i c_i$, where λ_i are binary weights for the i-th cell in the chain. ∂ is the boundary operator for finding the linear combination along the perimeter of a face or the 1-dimensional chain of a 2-cell. In Figure 4.3b, the perimeter is formed of edges a, b, c, d, e. The 1-dimensional chain of the face σ assuming $\lambda_i = 1$ is: $\partial\sigma = a + b + c + d + e$.

Discrete differential 1-forms (denoted as differential forms) function on 1-cells or edges. We use differential forms to keep track of the number of moving objects on each edge. Formally, a differential form is a function: $\xi : E \to \mathbb{R}$. It has a property where $\xi(-e) = -\xi(e)$, and we denote the differential form at time t as ξ_t. We find the number of moving objects contained in each face by integrating the differential forms along the 1-chain C: $\xi(C) = \sum_{e \in C} \xi(e)$.

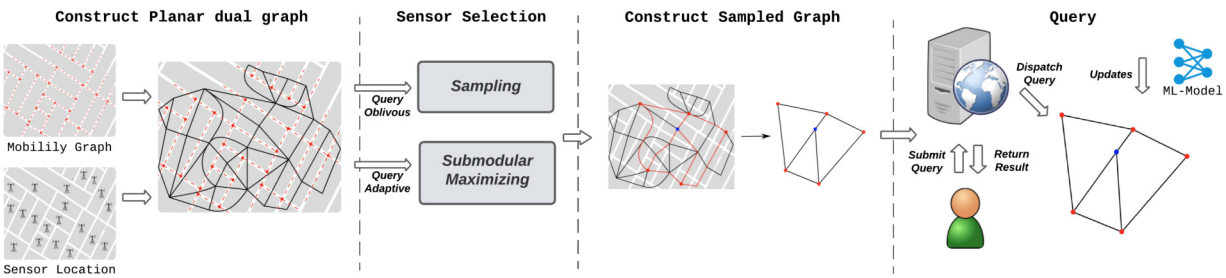


Figure 4.4: An overview of the framework.

4.4 Privacy-aware Range Query Framework

4.4.1 Overview

In this section, we present our framework for efficiently answering range count queries using planar graph representations. We first provide an overview in this subsection, shown in Figure 4.4, before breaking down each step in detail.

The first step is to generate a sensor configuration from a subset of sensors in an existing planar graph representation of the sensing and mobility domains. For completeness, we show how to construct a planar graph representation when the sensor and mobility graphs are unknown in Section 4.4.2, which may be useful for planning sensor development or using the sensor graph as a spatial decomposition technique. Once the planar graph representations are generated, we propose two approaches for selecting the sensors based on the amount of prior knowledge about the queries. If the query distribution is unknown or dynamic, we elect a subset of sensors using sampling algorithms. We showcase multiple sampling algorithms for different use cases in Section 4.4.3. When the query distribution is known, we elect the sensors based on a submodular maximization, where we maximize the sensor coverage given a budget. We present the submodular maximization algorithm in Section 4.4.4. The selected sensors will be the communication sensors. They also manage and aggregate counts from local sensors within their sensing region. This is often done in practice to limit long-distance communication (sensor-to-server) to a few sensors. An example can be found in [202].

After generating the subset of sensors, we construct a new "sampled" planar graph representation of the sensing domain and generate the edges between the nodes. For the query oblivious case, we connect the nodes either with a triangulation-based or kNN-based algorithm. The edges are then materialized in the network by routing through the shortest paths between selected nodes in the original sensor graph. We show the detailed steps in in Section 4.4.5.

Once the sample sensor graph is built, we enter the query stage. We first explain how queries are dispatched to relevant node(s) and provide the steps to find all nodes that boundary the

spatial query region in Section 4.4.6. We then show how to update the sampled graph without identifiers in Section 4.4.7 before presenting the query algorithms in Section 4.4.8. Lastly, we present the regression models used to store arbitrary timestamps in Section 4.4.9. As mentioned in Section 4.3.2, our goal is to improve query efficiency. Therefore, we refrain from detailed privacy analysis, but one can extend our method using methods from [83] to include privacy guarantees.

4.4.2 Constructing Planar Graphs

We now describe the process of generating planar mobility graphs from a spatial map or spatial network graph. For example, we can generate a road network graph from a standard map using mapping services. The first step is to filter out non-relevant nodes and edges, such as walking paths and train tracks. We then generate the planarized graph by removing intersections from underpasses and flyovers. A virtual sensor graph can be generated as the dual graph of the planarized mobility graph.

For road networks, moving objects travel along defined paths, constraining their movement to predefined paths. However, there are many cases where moving objects can freely roam around the spatial region (e.g., air and sea transportation). In this case, we may want to generate "virtual" paths from historical mobility traces. E.g., we generate paths from historical data where there is considerable traffic. However, estimating which paths to generate is not trivial; we will leave this for future work.

4.4.3 Query Oblivious Sensor Selection

In this section, we describe the sensor selection algorithms when we have no prior knowledge about the queries or when the query distribution is unknown. Formally, there are $|V|$ possible locations to place sensors, and the system has a budget for placing m sensors that optimizes query performance. We propose sampling to select a subset of nodes $\tilde{V} \subset V$ from the sensing

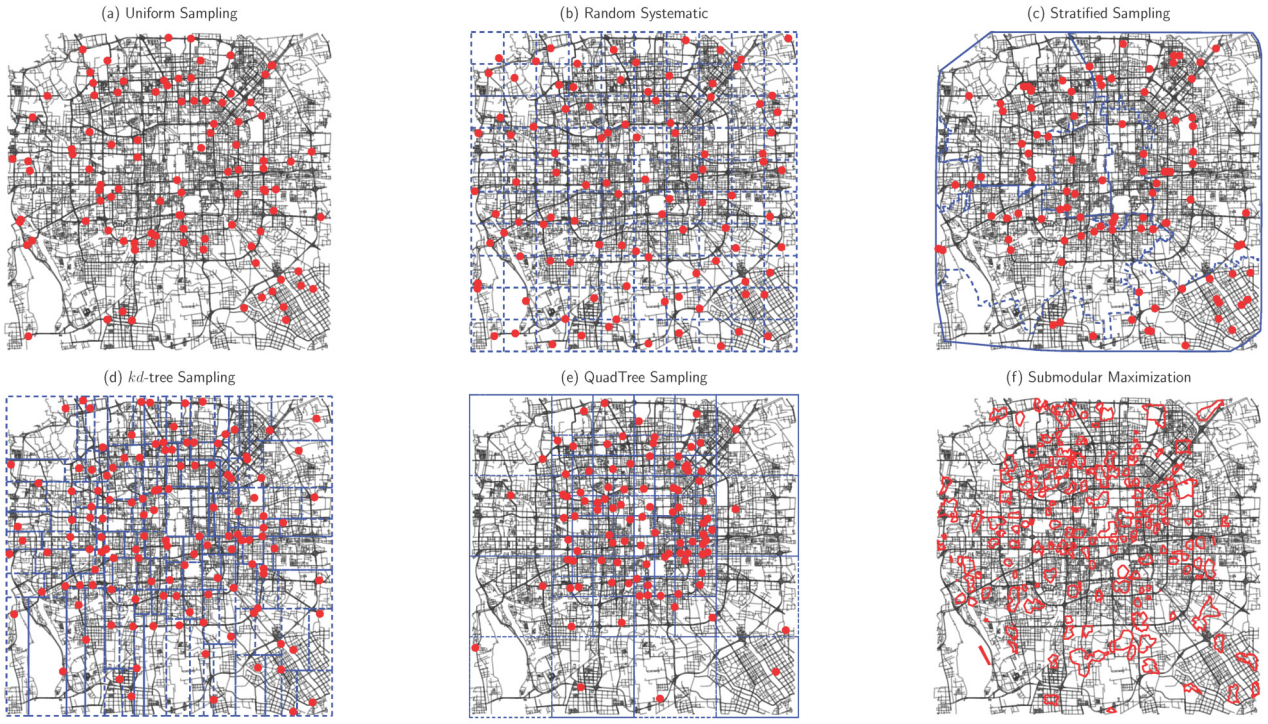


Figure 4.5: Sampling communication sensor location using road network of Beijing. (a) uniform sampling. (b) systematic sampling. (c) stratified sampling. (d) *kd*-tree sampling. (e) quadtree sampling. (f) Regions (red) selected by submodular maximization given a set of queries.

graph G to generate a sampled graph $\tilde{G}$. We now present sampling approaches aimed at various applications.

Uniform random sampling: we selects m nodes from V with equal probability. Figure 4.5a shows sensors selected using uniform sampling. This is the simplest type of sampling and has a bias towards denser regions. This method is suitable for applications that query denser areas. E.g., traffic monitoring applications may prefer more sensors in densely populated areas.

Systematic sampling: We impose a virtual grid and select a node from each grid cell. The chosen node is either closest to the cell center or is randomly selected in each cell. An example is shown in Figure 4.5b. Systematic sampling ensures a uniform distribution of nodes across the spatial domain. This is suitable for applications where queries are uniformly distributed in space.

Stratified sampling: We partition the mobility graph using strata (e.g., each district or zip-code region). We then uniformly sample a set of nodes from each stratum. Figure 4.5c shows

an example where strata are districts in Beijing. A function defines the number of samples in each stratum. For simplicity, we use a function that defines the number of samples based on the area of each stratum. This function can be easily extended to incorporate other factors, such as the importance of each stratum. This type of sampling is useful when queries are defined based on strata.

Hierarchical space partition-based sampling: We build a quadtree or a k-d tree for the nodes V in graph G by recursively partitioning the space until the leaf level has m nodes. Like systematic sampling, we select nodes closest to the center or randomly select nodes in each leaf node to form the sampled nodes. Figure 4.5d,e show the *kd*-tree and quadtree sampling methods where we randomly selected a node from the leaf nodes. This method benefits from both uniform and systematic sampling.

Note that we assume the cost of selecting any sensor to be uniform for the algorithms mentioned above. We can include non-uniformity by using different weights for each node. For example, if we were to make our sampling methods query adaptive, we could use the number of times each node appeared in previous queries as the weight.

4.4.4 Query Adaptive Sensor Selection

This section considers the case when the expected query regions are known a priori (e.g., from domain experts or predictions based on historical data). Similar to Section 4.4.3, we select m sensors from $|V|$ to form a subset of nodes $\tilde{V} \subset V \in G$ for generating the sampled graph $\tilde{G}$. We formulated the problem as a sensor placement problem and used submodular maximization [114] to approximate the solution. We start with how to use submodular maximization for sensor placement and adapt it to our problem accounting for query distribution.

Submodular maximization for selecting sensors: We start by defining the classical sensor placement problem based on coverage. There are n possible locations ($\mathcal{V}$) to place the sensors where each sensor has a defined region it can sense. Given that we have the budget to deploy m sensors, how do we choose the locations to maximize the total coverage area? This problem

is NP-hard [112] as all combinations of $\mathcal{V}$ need to be tested to choose the best m locations, which requires an exponential runtime.

We consider the submodular maximization approach. We denote the set of m locations as L. For simplicity, each sensor covers a unit circle in space. In this case, the utility function, which is the sensor coverage area, is submodular, giving it a property of diminishing returns. This means that the marginal area covered by each new sensor will become smaller with more sequentially added sensors. The reason is simply because new sensors are more likely to have a coverage area that overlaps with the existing set of sensors.

In terms of a submodular optimization problem, we consider a set of sensors S with a coverage of $f(S)$. The marginal coverage of a new sensor x, where $(x \notin S)$, is $\Delta_S f(x) = f(S \cup \{x\}) - f(S)$ with respect to S. In more general terms, for two sets of sensors A and B, where $A \subseteq B \subseteq \mathcal{V}$, the marginal coverage is:

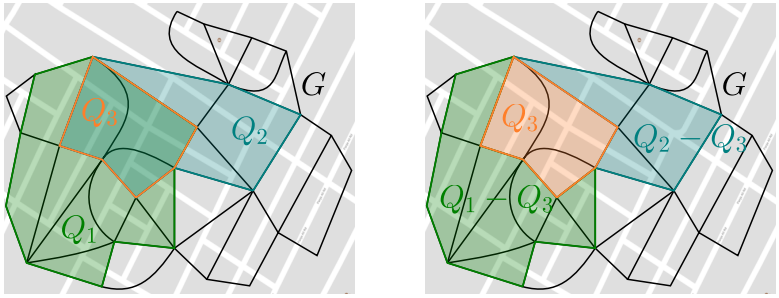
$$\Delta_A(x) \geq \Delta_B(x), \text{ or } f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$$

To maximize the sensor coverage, we can maximize the objective function, assuming the cost of selecting any sensor to be uniform as the following.

$$L = \text{argmax}_{L \subseteq \mathcal{V}} f(L) \text{ where, } |L| \leq m \tag{4.1}$$

We use an iterative greedy approximate solution to solve this maximization by starting with an empty set. At each iteration i, we choose the sensor with the maximum marginal utility (marginal coverage) until m sensors are selected. If two sensors have the same marginal utility, we use random selection. We can use the following equation to select a sensor at each iteration:

$$S_{i+1} = S_i \cup \text{argmax}_{v \in \mathcal{V} - S_i \text{ and } |S_i| < m} \Delta_{S_i}(v) \tag{4.2}$$



(a) (b)

Figure 4.6: Example of two overlapping query regions. (a) Q_1 and Q_2 are two overlapping query regions to form $Q_3 = Q_1 \cap Q_2$. (b) Disjoint query regions $Q_1 - Q_3$, $Q_2 - Q_3$ and Q_3.

We can account for non-uniform costs by adding another constraint based on a more general optimization budget notion to this problem.

$$L = \text{argmax}_{L \subseteq \mathcal{V}} f(L) \text{ where, } \sum_{v \in L} c(v) \leq B \tag{4.3}$$

Where v is a sensor, $c(v)$ is the cost function of selecting sensor v, and B is the optimization budget. We can update Equation 4.2 to the following.

$$S_{i+1} = S_i \cup \text{argmax}_{v \in \mathcal{V} - S_i \text{ and } c(v) \leq B - c(S_i)} \frac{\Delta_{S_i}(v)}{c(v)} \tag{4.4}$$

To use submodular maximization based on Equation 4.2 or 4.4, we can now adapt the utility function to account for query distributions rather than sensor coverage.

Submodular utility function given query regions: We consider a set of historical query regions, $\bar{\mathcal{Q}}_R = \{\bar{Q}_R\}$ and change our objective to find the query regions to monitor by placing the sensors at the boundaries of these regions.

Now, the problem is based on monitoring the boundaries of the regions (which differs from the coverage problem above). Before selecting any query regions, we have to consider the problem of overlapping boundaries. Let us consider two regions Q_1 and Q_2 in Figure 4.6a. They overlap to form an overlapping region $Q_3 = Q_1 \cap Q_2$. If we select the edges of Q_1 or ∂Q_1, the set of sensors cannot answer any query about Q_2. However, if we also add ∂Q_3, we can answer

(lower-bound) queries in both Q_1 and Q_2 with a small additional cost (since $|\partial Q_3 \cap \partial Q_1| > 0$). Alternatively, we can minimize the cost by only selecting ∂Q_3. Therefore, we preprocess $\bar{\mathcal{Q}}_R$ by maximally partitioning it into a set of non-overlapping query regions $\mathcal{Q}_R = \{Q_R\}$. Figure 4.6b shows the resulting query regions.

We can now formulate our utility and cost functions. We first denote σ as a maximal cell complex fully contained by at least one query region in $\mathcal{Q}_R$, which the sensor monitors along its perimeter $\partial\sigma$. Assuming the cost of each edge is uniform, the cost function $c(\sigma)$ for selecting the cell complex σ is:

$$c(\sigma) = |\partial\sigma| \tag{4.5}$$

For simplicity, we set $c(\sigma)$ as the number of boundary edges. We can use a more complex function to incorporate the actual cost as long as it monotonically increases with edges (e.g., any function where increasing the number of edges increases the cost). We formalize the utility function $f(\sigma)$ for selecting the cell complex σ as:

$$f(\sigma) = \sum_{\forall Q_R \text{ containing } \sigma} \frac{\omega(\sigma)}{\omega(Q_R)} \tag{4.6}$$

Where, $\omega(\sigma)$ denotes the number of cells within σ. The utility of the region σ increases if it covers a larger area w.r.t a query region in $\mathcal{Q}_R$. The utility function $f(\sigma)$ is always monotone and submodular since (1) the ratio $\frac{\omega(S)}{\omega(Q_R)}$ never decreases with more sensors added to S. (2) $\omega(.)$ is submodular since we deal with an area. To select the subset of nodes $\tilde{V}$, we first select the sensing region σ that maximizes the utility function $f(\sigma)$ until the budget m. Figure 4.5f shows an example of selected regions. We can replace $\omega(\sigma)$ with a more general notion representing cells with different weights.

Note that our formulation of Equation 4.6 is for lower bound approximations (the resulting count will be less or equal to the actual count). For upper bound approximations, we can modify the σ to denote a cell complex that contains one query region in $\mathcal{Q}_R$.

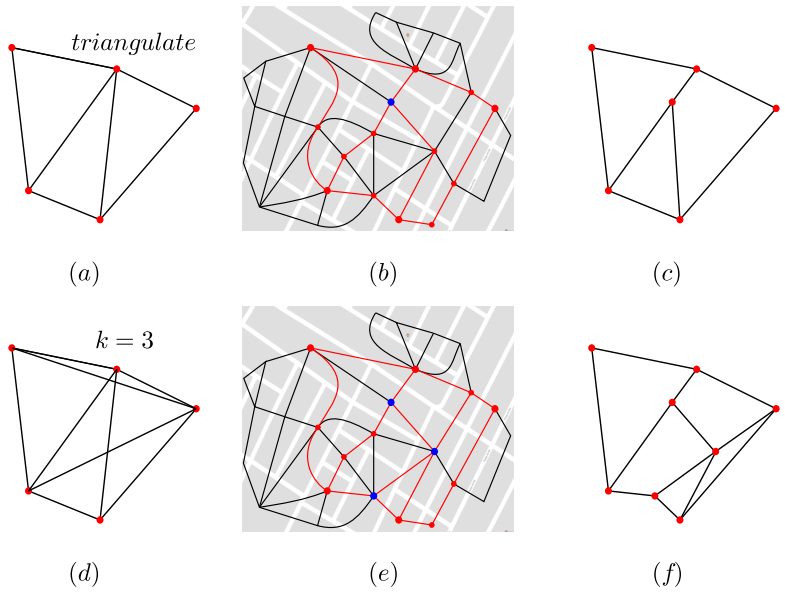


Figure 4.7: Generating edges from sampled nodes using triangulation and kNN (k=3). (a) Edges after triangulating $\tilde{V}$. (d) Edges after finding k-nearest neighbor for each node v in $\tilde{V}$. (b,e) Replacing edges within the shortest path in G. The blue nodes are intersection nodes added to ensure $\tilde{G}$ is planar. (c,f) $\tilde{G}$ after simplification.

4.4.5 Generating the Sampled Graph

Once the subset of nodes $\tilde{V}$ is found, we generate the sampled graph by connecting the nodes with edges. For the query oblivious selection method, we first need to generate the edges between the nodes, either using triangulation or kNN.

Figure 4.7a shows how triangulated edges are formed from $\tilde{V}$ (red), and Figure 4.7d shows edges formed by kNN ($k = 3$). For kNN , we find the k-nearest neighbor nodes $\tilde{u} \in \mathcal{N}_k(\tilde{v})$ from $\tilde{V}$ for each node $\tilde{v} \in \tilde{V}$. kNN typically generates more faces, each with a smaller area. Note that $\tilde{G}$ becomes maximal when $k = m$.

Once the edges are formed between nodes, we must map the edges to network paths to avoid any intersections (which makes the graph non-planar). This is formalized by routing the shortest paths between nodes in the original sensing graph G. Figure 4.7b & e shows the shortest paths, where intersections are labeled as blue nodes. We add the intersections to the sampled graph for virtual representation purposes (they do not have to be communication sensors). The resultant sampled graph $\tilde{G}$ is shown in Figure 4.7c & f.

For the submodular maximization method, edges are already formed from the selected regions.

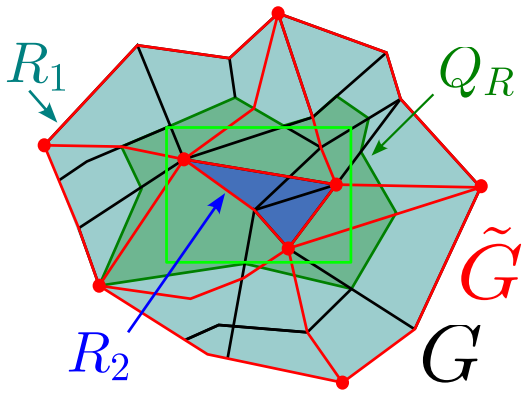


Figure 4.8: The query region Q_R (green) defines the rectangular query region as faces in the sensing graph G. R_1 and R_2 are the upper and lower approximation of Q_R.

We only have to map the edges to network paths via shortest paths in G. For example, we can use a case where triangles form each region. The first region contains three nodes, which will form a triangle. We can add another node to form another triangle with two nodes from the existing triangle for any neighboring regions. If the regions are disjoint, we connect them via the shortest path between nodes to form an enclosed area. The process continues until we have m nodes.

4.4.6 Dispatching Queries

We now enter the query stage. A query is dispatched from a remote query server to the communication nodes in the sampled graph $\tilde{G}$. Before explaining which nodes are dispatched to, we first introduce how the graphs define the query regions. The remote query server knows the structure of sensing graph G and sampled graph $\tilde{G}$ but does not know the exact counts (counts are stored locally at the sensors). The query region is defined as the union faces of the sensing graph G, which supports the query region of any arbitrary shape. E.g., we enclose Q_R to a rectangular query region in Figure 4.8.

Since $\tilde{G}$ is a subgraph of G, the queries answered in $\tilde{G}$ will be approximate as we cannot guarantee Q_R to be present in $\tilde{G}$. There are two approximations: lower-bound approximation, which is the maximal region enclosed by the query region Q_R (shown as R_2), and upper-bound approximation, which is the minimal region containing Q_R (shown as R_1). Depending on the

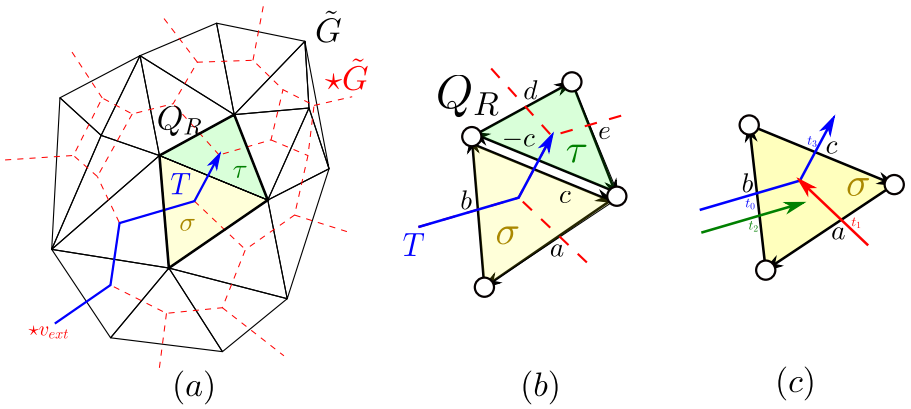


Figure 4.9: Differential form example. (a) The road network is the dual graph (dotted line), with the sensor network as the primal graph (solid line). $\star v_{ext}$ is the infinity node that acts as a source and sink for objects entering and leaving the mobility graph. (b) Target T moving from face σ to τ. The query region Q_R is the union of faces σ and τ ($Q_R = \sigma \cup \tau$). (c) An alternative example of three trajectories moving in and out of σ at different timestamps.

type of approximation (configurable by the user), we define the query region as the union of faces in $\tilde{G}$ that are bounded by (lower-bound) or bounds (upper-bound) the query region.

Once the query region is defined in $\tilde{G}$, there are two ways to communicate with the sensors depending on communication cost. The first method involves communicating with all nodes on the query region's perimeter and aggregating the final count in the query server. The second method involves communicating with one node on the perimeter of the query region. The node then finds all other nodes on the perimeter via node traversal (e.g., flooding, routing, or combination [39]) and aggregates the count back to the node before returning the result to the query server. The choice of method depends on the actual cost in the network and is not the focus of our framework.

4.4.7 Updating the Sampled Graph

Snapshot: We first show how the differential forms are updated without timestamps (or a snapshot in time) before extending it to include timestamps. The idea is to have a differential form along each edge $\tilde{e}$ in $\tilde{G}$ to keep track of the count (physically, the count is stored at the nodes, and the edge counts act as an abstraction). Here, we can take advantage of the vertex-edge duality, where the object moves along the edge $\star\tilde{e}$ in $\star\tilde{G}$ will cross an edge $\tilde{e}$ in $\tilde{G}$.

Therefore, the corresponding differential form $\xi(\tilde{e})$ will be updated.

We define two tracking forms for each directed edge: $\xi_t^+(.)$ and $\xi_t^-(.)$. An example is shown in Figure 4.9b where object T moves from source (σ) to target (τ). The shared edge crossed by T is seen as c for $\partial\sigma$ and $-c$ for $\partial\tau$. Therefore, T leaves σ via directed edge c in the perspective of $\partial\sigma$. We use $\xi_t^-(c)$ to track this type of movement. Similarly, we use $\xi_t^+(-c)$ to track T as it enters τ via $-c$ in the perspective of $\partial\tau$. We use $+$ to denote the movement from one face to another:

$$\xi_t^+(-c) = \xi_t^+(-c) + 1 \text{ and } \xi_t^-(c) = \xi_t^-(c) + 1 \tag{4.7}$$

Theorem 4.1 *If the differential forms are updated as per Equation 4.7, then at the current time t, the number of objects inside a cell complex Q_R is given by $\sum_{e\in\partial Q} \xi_t^+(e) - \xi_t^-(e)$.*

Proof sketch. In Figure 4.9a & b, τ is initially empty ($\partial(\tau) = 0$). Target T enters τ at time t from σ and crosses at edge c. According to Equation 4.7, $\xi_t^-(c) = 1$ since T is moving out of σ. Similarly, $\xi_t^+(-c) = 1$ as T moves into τ. We find the count by integrating the differential forms along the perimeter of the face. The count at time t is $\xi_t(\partial(\tau)) = (\xi_t^+(d) - \xi_t^-(d)) + (\xi_t^+(e) - \xi_t^-(e)) + (\xi_t^+(-c) - \xi_t^-(-c)) = \xi_t^+(-c) = 1$. Similarly, the count in σ is $\xi_t(\partial(\sigma)) = \xi_t^+(b) - \xi_t^-(c) = 1 - 1 = 0$.

Arbitrary Time Intervals: We extend the tracking form to contain a sequence of timestamps for each crossing event to answer spatiotemporal range queries. We use $\oplus$ to represent appending to a sequence (E.g., $\vec{x} \oplus y$ means appending y to a sequence $\vec{x}$). Thus, when an object crosses an edge c in Figure 4.9, we update the tracking forms as follows.

$$\gamma_t^+(-c) = \gamma_{t-1}^+(-c) \oplus t \text{ and } \gamma_t^-(c) = \gamma_{t-1}^-(c) \oplus t \tag{4.8}$$

4.4.8 Querying the Sampled Graph

Static object count: We use a function $C(\gamma_t(e), t)$ to count the number of crossing events from $-\infty$ to t at edge e. We answer the static object count query by combining Equation 4.8, $C(\gamma_t(e), t)$ and Theorem 4.1.

Theorem 4.2 *If the tracking forms are updated as per Equation 4.8, the number of objects inside a cell complex σ up until t_q, is given by: $\gamma_t(\partial\sigma) = \sum_{e\in\partial\sigma} C(\gamma_t^+(e), t_q) - C(\gamma_t^-(e), t_q)$.*

Proof sketch. In Figure 4.9c, a blue trajectory enters through edge b at t_0 and exits from edge c at t_3. The green trajectory enters through edge b at t_2, and the red trajectory enters through edge a at t_1. According to Equation 4.8, $\gamma_{t_3}^+(a) = \{t_1\}$ since the red trajectories enters through edge a at t_1. Similarly,$\gamma_{t_3}^+(b) = \{t_0, t_2\}$ and $\gamma_{t_3}^-(c) = \{t_3\}$. We find the count in σ by applying Theorem 4.2. Therefore, the count is $\gamma_t(\partial\sigma) = C(\gamma_t^+(a), t_3) + C(\gamma_t^+(b), t_3) - C(\gamma_t^-(c), t_3) = 1 + 2 - 1 = 2$.

Transient object count: We modify the count function to find the number of objects between an arbitrary time range for the transient object count case. The count function takes the form $C(\gamma_t(e), t_1, t_2) = C(\gamma_t(e), t_2) - C(\gamma_t(e), t_1)$, where it returns the number of events from t_1 until t_2 at an edge e.

Theorem 4.3 *The number of objects that entered and left a cell complex σ from t_0 to t_1 is given by: $\gamma_t(\partial\sigma) = \sum_{e\in\partial\sigma} C(\gamma_t^+(e), t_0, t_1) - C(\gamma_t^-(e), t_0, t_1)$, where negative results mean more moving objects left during the time interval and vice versa.*

Proof sketch. In Figure 4.9c, we want to find the transient object count in σ between $[t_1, t_3]$. The count at edge b is $C(\gamma_t^+(b), t_1, t_3) = C(\gamma_t^+(b), t_3) - C(\gamma_t^+(b), t_1) = 2 - 1 = 1$. The same logic is applied to edges a and c. According to Theorem 4.3, the count is $\gamma_t(\partial\sigma) = C(\gamma_t^+(a), t_1, t_3) + C(\gamma_t^+(b), t_1, t_3) - C(\gamma_t^-(c), t_1, t_3) = 0 + 1 - 1 = 0$. Intuitively, this makes sense as the red trajectory enters at t_1 and the blue trajectory leaves at t_3, which cancel each other out.

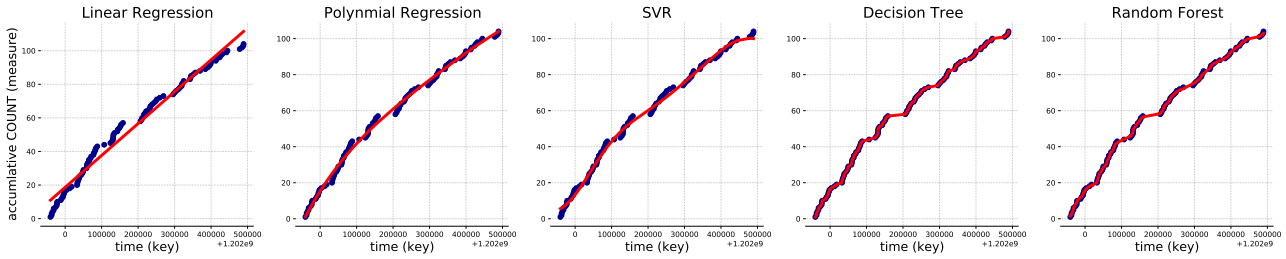


Figure 4.10: Storing the range count and timestamp(s) in an example tracking form using popular regressors.

4.4.9 Computing the count function using regression models

Lastly, we introduce machine-learning models that efficiently store and process timestamp sequences. The naive method to compute $C(\gamma_t(e), t)$ is to search the count at time t in the sequence $\gamma_t(e)$ at edge e. We can use indexing methods such as the B-tree to efficiently search and store the sequence. However, such indexing methods require the sequence to be stored explicitly, thus increasing the storage size with increasing data size. We take inspiration from learned indexes to compact the sequence of timestamps, which saves space. The tracking form's monotonic property (the temporal dimension continuously increases and γ^+ and γ^- always increase) allows us to use a similar approach to store tracking forms at each edge. In this case, we only store the model parameters and infer the results without any last-mile search. This is fine for our framework since it is based on approximations. However, if one requests exact results, learned indexes would be needed, which we present in Chapter 5.

Specifically, we store the timestamps as a CDF (cumulative distribution function) and predict the cumulative count of events that happened till a given timestamp t. The count is approximated by using the predicted count as the final answer. The lookup time is reduced to $O(1)$ (considering the model does not increase its parameters as we receive more events). Simple regression models are used to reduce storage and training costs. Figure 4.10 shows popular regressors modeling the function $C(\gamma_t(e), t)$ at an edge e. Assuming we are using linear regression (Figure 4.10a), we can compute the count $C(\gamma_t(e), t)$ by applying $C(\gamma_t(e), t) = \alpha + \beta t$, where α and β are the intercept and slope of the linear model. The range count $C(\gamma_t(e), t_0, t_1)$ is the difference between $C(\gamma_t(e), t_1) = \alpha + \beta t_1$ and $C(\gamma_t(e), t_0) = \alpha + \beta t_0$.

We use a limited-size buffer to store new crossing events for updates (accommodate n events). When the buffer is full, we build a new model and flush the buffer to make space for updates. When combing the model and the buffer, we can answer range queries on at most $2n$ events in the past. We can further reduce the storage space by learning the regressors incrementally. For example, we learn a model at time t that combines the buffer $[t-n, t]$ and the trained model at $t-n$. Alternatively, without storing the keys, we can use a similar queue-type learned index structure of FLIRT (Chapter 5).

4.5 Experimental Evaluation

In our experiments, we simulate an in-network system with abstractions. The main focus is evaluating the algorithmic improvement independent of the real distributed implementation (e.g., the actual communication protocol). The evaluation examines our framework's performance against the baseline for static and transient count queries under lower bound & upper bound approximation, edge generation methods (kNN vs. triangulation), query misses, and systems gains. Highlights of the results are summarized as the following:

- We show significant performance gains in terms of processing speed, the number of nodes accessed, and storage size compared to the baseline while reducing relative error.
- The submodular maximization method achieves the lowest relative error.
- The *kd*-tree and the quadtree have the lowest relative error for sampling methods.
- kNN based connectivity has lower relative error for smaller query regions.
- The regression models provide speedup in query execution time and reduction in storage requirement with minimal impact on relative error.

4.5.1 Experimental setup

Dataset & Setup: The mobility graph uses a real-world road network of Beijing [1] shown in Figure 4.5. The moving objects data uses two GPS trajectory datasets, T-drive [207] and Geolife [217]. The combined dataset contains 29,027 trajectories collected from February 2007 to August 2012.

We conduct the experiments on a machine with Intel E5-2680 v3@2.50GHz running Centos 8. The core count is 48, and the memory size is 128GB. Each experiment is repeated 50 times. We report the median of the runs in solid lines, and the shaded regions are between the 25th and 75th percentiles.

Baseline: The obvious baseline compares our method against an unsampled sensing graph G based on [157] (Does not support arbitrary time intervals). However, we also want to compare with a baseline that considers existing works on privacy-aware aggregate range queries and sampling. The baseline combines these methods, as no single baseline meets all our goals.

Specifically, we combine methods from privacy-aware aggregate range queries [70, 82], in-network query processing [157] and trajectory sampling [122, 59]. To simulate tracking in a sensor network, we use [157] to build a graph over the mobility and sensing domain. We then use the distributed Euler-histograms [70, 82] to count the number of objects within each face of the graph rather than keeping counts along the edges. This method is designed for centralized systems. Therefore, here, we assume that all counts are aggregated and stored in the nodes before querying (which the method itself does not support). To incorporate sampling a subset of sensors, we use a random index sampling algorithm based on [122, 59] to uniformly sample faces in the graph instead of edges as in our framework.

Preprocessing: The road network is discretized using [31]. We insert nodes at any edge intersections and remove contour nodes (nodes that are not junctions added by maps to indicate the geometry of roads). We then map-match the trajectories to the road network by mapping each trajectory location to the nearest node and connecting them via the shortest path in the graph.

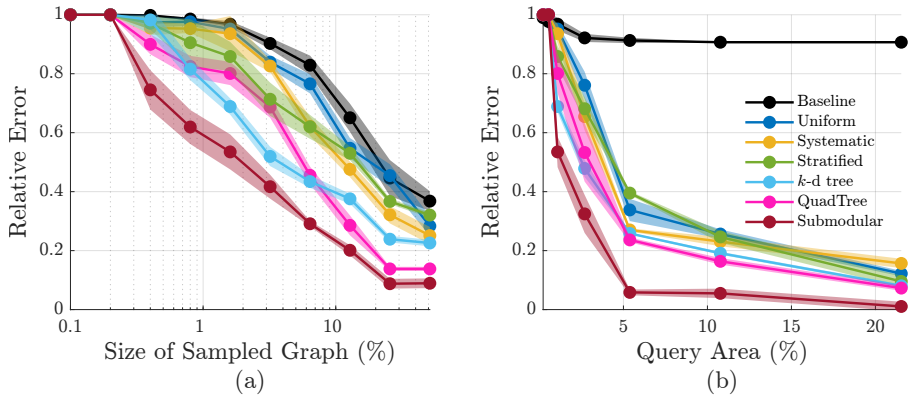


Figure 4.11: Lower bound error of static queries. *(a)* w.r.t graph size *(b)* w.r.t query sizes.

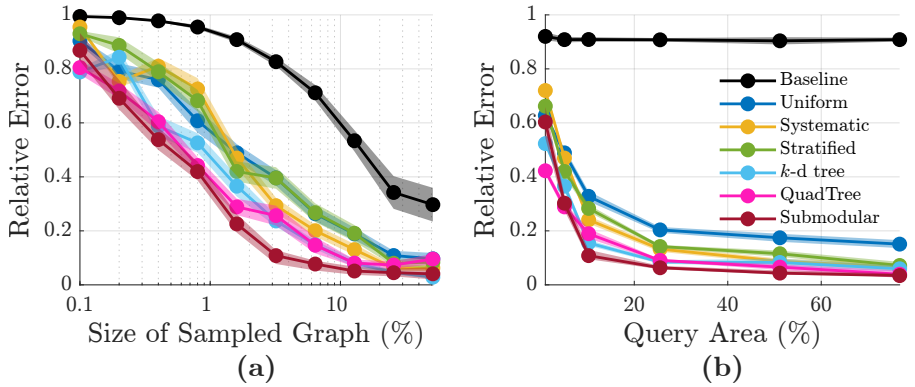


Figure 4.12: Lower bound error of transient queries *(a)* w.r.t graph size. *(b)* w.r.t query sizes.

Performance measures: We use relative error to evaluate the estimation error from approximation. The relative error is $\frac{|\eta-\hat{\eta}|}{\eta}$ where η denotes the actual range count (count from the unsampled graph G) and $\hat{\eta}$ denotes the approximated range counts. The closer the relative error is to zero, the more accurate the approximation.

Query generation: As mentioned in Section 4.4.6, the queries are defined in terms of the sensor network to communicate with all relevant sensors for the query. Therefore, we define query region as a union of faces of the non-sampled sensing graph G.

We simulate this by selecting a rectangular spatial region of area A before finding the faces contained by the rectangular region in the sensing graph G. The union of the faces forms a subgraph, which becomes our query region Q_R. The resulting query regions will have irregular shapes with an upper bound area of A. The temporal range is randomly sampled 7-day periods. Additionally, we choose 100 query regions uniformly as the historical data for generating sensor placements with the submodular maximization algorithm.

4.5.2 Size of the Sampled Graph

We evaluate the frameworks under varying sizes of the sampled graph. Figure 4.11a for the static object count and Figure 4.12a for the transient object count. The queries executed are lower bound approximations, and the query regions are fixed at 1.08% ($10km^2$ in km), defined as the percentage of the total spatial area. The sizes are shown as a percentage with respect to the original sensing graph G. Our sampling methods outperform the baselines, with kd-tree and quadtree having the lowest error. The submodular maximization method further decreases the error by considering historical query distribution.

Our method can cover more area than the baseline, which scatters sampled faces across the region. We consider the sensor distribution and track moving objects based on edge movements rather than keeping counts based on faces. Therefore, we can achieve a lower error for the same size (if we have vertical lines on the graph). There is a lower limit to the relative error we can achieve due to the inherent nature of approximations, shown as the plateau at larger sizes. The baseline also has a lower limit, but it requires more samples to reach, which falls outside the scope of these figures. The highlight of our method is its ability to reduce the number of samples for the same relative error.

Interestingly, all methods were shown to have reasonably low errors at the sampled size of 25%. However, in the $0.4\% - 3.2\%$ range, all methods show poor performance, suggesting that sampling is a viable approach if the sample size is sufficient.

4.5.3 Query Region Size

Next, we show the relative error against varying query region sizes. The area of the query region is shown as the percentage of the total sensing area. Figure 4.11b shows the static object count, and Figure 4.12b for the transient object count over an extended query range. The size of the sampled graph is kept constant at the median graph size of 6%.

The relative error decreases with increasing query regions as the likelihood of the query region

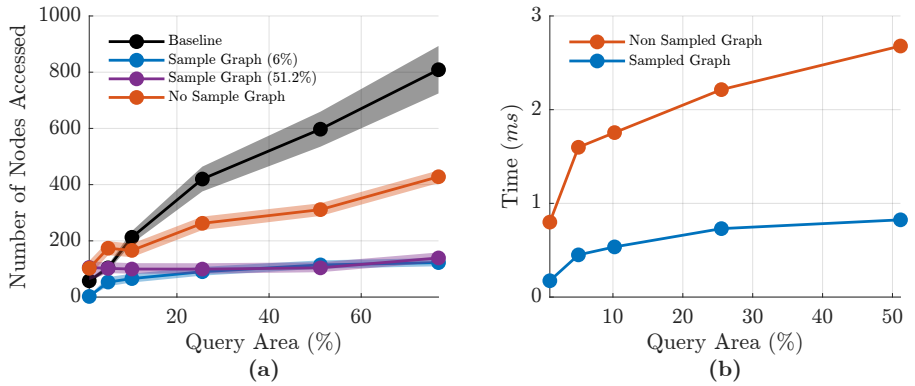


Figure 4.13: *(a)* The number of nodes accessed w.r.t query sizes. *(b)* Execution time.

containing a sampled face increases, with the submodular maximization method scaling exceptionally well with query size. This heavily impacts the effectiveness of the sampling methods, as smaller query regions are less likely to be contained by any sampled regions, which generates substantial errors. As we shall see in Section 4.5.5, the sampled graph does not cover enough area to answer small-area queries. On the other hand, the baseline only depends on the number of samples, as the area of the sampled faces predetermines the maximum coverage.

4.5.4 Communication Cost & Speed up

We evaluate the communication cost with the number of nodes accessed for the sampled graph, unsampled graph, and baseline in Figure 4.13a. We show results with a sampled graph size of 6% and 51.2%. Note that the larger query sizes do not necessarily mean more nodes are accessed (although they are correlated). E.g., in Figure 4.7c, the number of nodes accessed is five if the query region covers the entire perimeter or only the left triangle.

We show that sampled graphs achieve a logarithmic shape that plateaus at larger query areas. The number of node access is very low for small areas, which also explains the high relative error. On the other hand, the number of accesses for the baseline and the unsampled graph linearly increases with the query area as we need to access all nodes within the query region, showing a clear correlation.

Even though the node access is near constant for larger query sizes, we show that the average querying time increases with query size in Figure 4.13b. This is because the planar graph

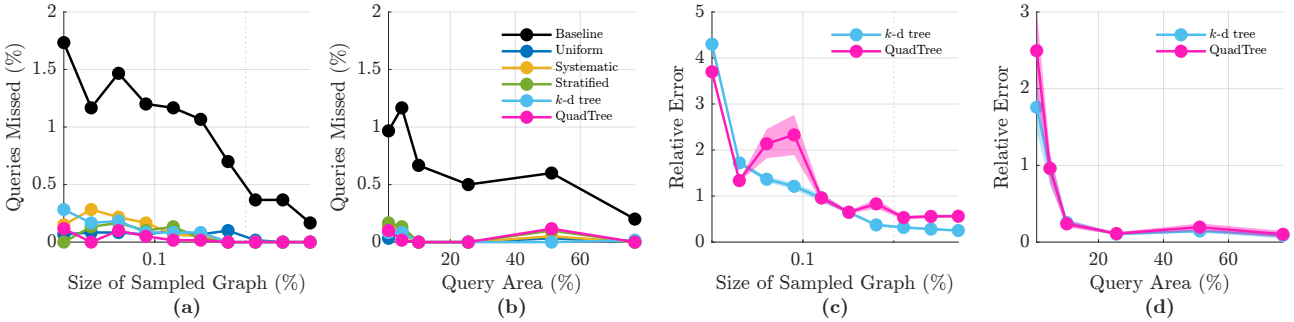


Figure 4.14: *(a,b)* The number of queries missed over the total number of queries. *(c,d)* The upper bound relative error.

represents an abstraction of the underlying sensor network. Data has to be aggregated to the sampled nodes. Larger query regions mean longer distances between the nodes, and the aggregation may need to route through more nodes. However, the sampled graph does reduce the processing time of queries, as shown by the shallower slope in larger query regions.

4.5.5 Queries missed

We investigate the cause of the high relative error for smaller query regions and graph sizes by evaluating the number of queries missed. Query misses occur when the query region does not intersect with the sampled graph. Figure 4.14a & b shows the number of queries missed. The results show that query misses rarely occur for our method, which suggests the high relative error is mainly caused by the faces in the sampled graph being too large to be contained in the lower bound approximation.

4.5.6 Upper bound approximation

We then look at the relative error for upper bound queries in Figure 4.14c & d. The upper bound count is given by the minimal region $\tilde{G}$ that contains the query region Q_R. The count is larger or equal to the actual count since it contains objects not in the query region (hence, the y-axis is greater than 1). The results show a similar trend as the lower bound count – with increasing query region and size of $\tilde{G}$, the error reduces. Note that the submodular maximization is not shown since our formulation is for the lower bound approximation.

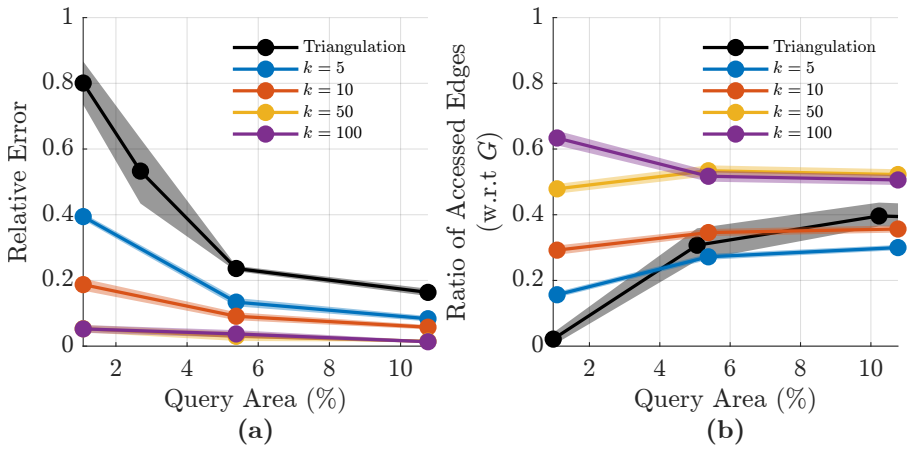


Figure 4.15: *(a)* The lower bound relative error of using kNN-based connectivity. *(b)* The number of edges accessed in the sampled graph w.r.t to G.

4.5.7 *k*NN based connectivity

We also investigate if generating edges using kNN after sampling would improve the error for smaller query areas since kNN typically generates many smaller faces. Figure 4.15a shows the relative error of using kNN compared to triangulation for sampling with a quadtree. The error decreases with increasing k for the same query region. However, if we look at Figure 4.15b, we see that the number of edges accessed increases, which means more nodes need to be communicated. $k = 5$ is shown to be able to decrease the error and have less edge access compared to the triangulation method. This suggests that having more faces, each with smaller areas, effectively reduces errors for smaller query areas but may be wasteful for larger query areas.

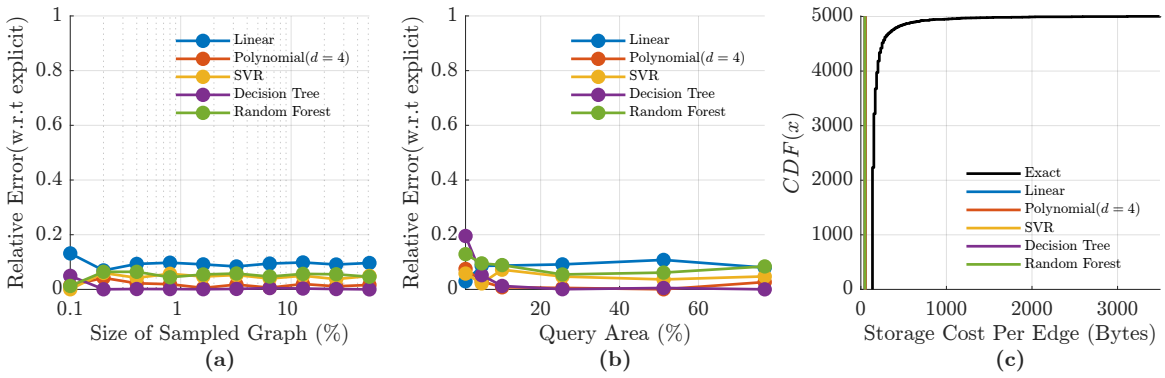


Figure 4.16: *(a,b)* The additional error generated by the regression models. *(c)* The storage cost of edges.

4.5.8 Performance of regression model

Lastly, we evaluate the error generated from the regression model and its space-saving. Figure 4.16a & b shows the error relative to storing counts explicitly (not to be confused with the count from unsampled graph G) for different regression models. The results suggest that simple regressors generate low error overhead. On average, the regression models add a 2.5% error penalty, greatly outweighed by the query speedup and storage reduction from these models.

Specifically, Figure 4.16c shows the storage-saving from the models. The y-axis shows the CDF of the number of timestamps stored in each edge with storage size in the x-axis. The CDF of the exact method shows there are more edges with smaller sizes and fewer edges with larger sizes. At around 500, there is a dramatic decrease in edges with storage costs > 500. On the other hand, the size of regression models is independent of the number of items on each edge. The maximum size of the learned sampled graph can be calculated by $n_{edges} \times size_{model} \times 2$ regardless of the number of targets. We do not consider updates to the regression model in this evaluation; we will focus on updates in the temporal dimensions in Chapters 5 & 6.

4.6 Summary

In this chapter, we present a framework to efficiently answer privacy-aware aggregate spatiotemporal range queries within the sensor network. A planar graph representation allows our framework to be aware of the sensor distribution. We propose techniques to select a subset of sensors to reduce the number of communications based on query distribution. Additionally, we support queries with arbitrary temporal ranges with regression models to accelerate the query performance and reduce storage.

Chapter 5

An Updatable Learned Index for Temporal Streams

5.1 Overview

From this chapter, we move to a local perspective and optimize the performance locally on each machine. From Chapter 3 & 4, we showed that updates cause the temporal dimension to continuously grow in contrast to updates in the spatial dimension being confined. Therefore, the characteristics of the temporal dimension are reminiscent of streaming data, which are naturally time-ordered and infinitely long [86]. We treat the indexing of the temporal dimension as a stream index problem and apply the idea of learned indexing to improve the lookup time and reduce storage size. Specifically, in this chapter, we explore the case where keys are in monotonic order (e.g., the tracking form in Chapter 4) before exploring a general case in Chapter 6.

Since streaming data can be infinitely long, we only maintain a finite sequence of the most recent data for practicality. This is known as a sliding window, where the window size determines the size of the sequence. The window size can be set using a specific count of records or a temporal threshold (e.g., one hour) to determine when records are considered outdated and removed. The challenge of handling streaming data is the ability to answer queries quickly under frequent updates. This calls for an index structure that is both read and write-efficient.

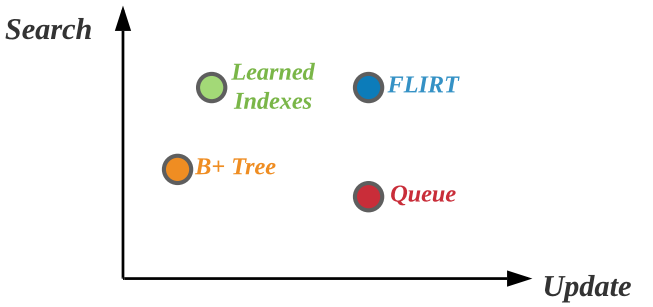


Figure 5.1: search vs update latency spectrum for sliding window processing.

Learned indexes have recently emerged as promising replacements for traditional indexes, especially when indexing static data. However, learned indexes that support dynamic data [72, 142, 93] are not designed for frequent updates, such as streaming data. The update performance suffers from expensive training costs of complex hierarchical structures designed for maintaining search accuracy. Specifically, they have to be constantly retrained and adjusted under a continuously updated set of keys where the distribution of the keys constantly drifts over time. We consider these updatable learned indexes as search-optimized data structures that "accommodate" updates, similar to algorithmic indexes such as the B+tree. Figure 5.1 shows where learned indexes and B+trees fall in the update and search performance spectrum.

Alternatively, we can use a more updatable data structure, such as a queue (implemented as an array or linked list). Although queues allow for efficient addition (enqueue) and removal (dequeue) of records, they are not designed for querying. A lookup requires a binary search over the entire sliding window, which is considerably slower than learned indexes [108, 128]. Therefore, queues fall on the opposite end of the spectrum compared to learned indexes and B+trees (shown in Figure 5.1). They are considered update-efficient data structures that support search. Therefore, we aim to synergize the update efficiency of queues with the predictive power of learned models to develop an index that handles both read-intensive and write-intensive workloads (the area that occupies the top right corner of Figure 5.1).

In this chapter, we introduce FLIRT (Fast Learned Index for Rolling Time-frames), a novel parameter-free index for sliding windows under continuous updates (insertions & deletes). FLIRT handles a workload where keys are monotonically ordered (e.g., time series and network packets). To enable efficient search and updates, we split the queue type structure into

data and root levels. At the data level, we implement Linearly Predictable Segments (LP-Segments) that use a linear model to accelerate a lookup and do not require any retraining or structural adjustment. The root level manages the LP-Segments with a circular queue structure to improve update performance.

Furthermore, FLIRT is designed to be self-optimized by actively adjusting itself to minimize search costs when the data distribution of keys evolves during continuous updates. Specifically, we achieve this by automatically adjusting the internal error threshold that controls the number of segments and the efficiency of searches within each LP-Segment.

Lastly, we further boost search speeds and accommodate distributed workloads in trajectory processing with two concurrent versions of FLIRT: ParallelPartitionedFlirt (PPFlirt) and ParallelSharedFlirt (PSFlirt). PPFlirt incorporates data partitioning for uniform query workloads, while PSFlirt distributes the queries for skewed query workloads.

5.2 Drawbacks of Existing Works

In this section, we first recap some indexes introduced in Section 2.3.3 & 2.4.4, before reviewing some streaming-specific indexes.

Recap of Generic Algorithmic and Learned Indexes: Most algorithmic indexes introduced in Section 2.3.3 can be used to index streams. However, as mentioned previously, they do not take advantage of data distribution to speed up queries and reduce storage size. On the other hand, the learned indexes introduced in Section 2.4.4, including ALEX [56], PGM-index [72], XIndex [180] and FINEdex [119], are all considered read optimized indexes that support updates.

Stream Indexes: We introduced the queue structure, which is the most basic type of stream index, in Section 2.3.3. A queue is typically implemented as a circular queue. [85] adapts the circular queue for different query types by employing a circular array of basic windows. Each basic window comprises a summary of keys aimed at reducing memory consumption and

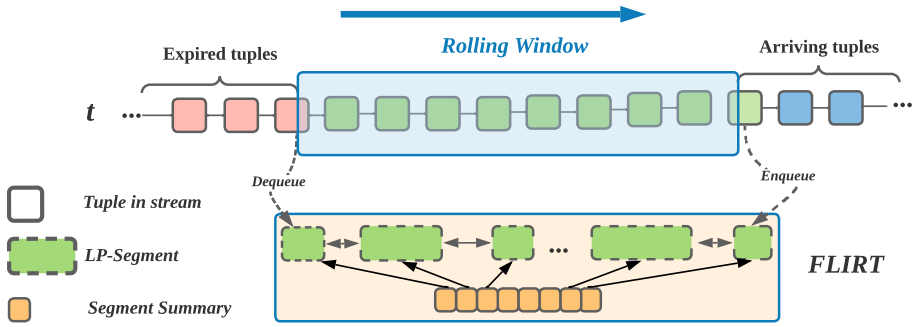


Figure 5.2: Overview of a monotonic key stream workload indexed by FLIRT.

improving query performance. The authors show for most scenarios that using linked lists as basic windows is preferred. They coined this variant LIST. We include LIST as one of our baselines and show that linked list structures are extremely inefficient for searching (due to excessive amounts of pointer traversals). Other works include introducing indexes for streaming joins via multiple B-trees [162] (details in Chaper 6).

5.3 A Fast Learned Index for Rolling Time-frames

5.3.1 Overview

For learned indexing, handling data streams involves managing continuously produced data that change the distribution over time. Furthermore, they need to handle the unique update patterns of monotonic keys, where insertions are at the end (enqueue) and deletions are at the beginning (dequeue) of the data sequence. For example, the timestamps of the trajectory during filtering are always time-sorted. Figure 5.2 shows this example with FLIRT, where new timestamps are appended to the end of the index, and expired timestamps are erased from the start. For the rest of the chapter, we denote this type of insertion (to end) and deletion (from beginning) as enqueues and dequeues, respectively.

As mentioned in Section 2.4.2, the search speedup of learned indexes originates from a machine-learning model approximating the mapping function between the keys and the physical storage location. FLIRT employs piecewise linear functions as the mapping function, which are inex-

pensive in terms of computation and storage cost [78, 109, 72, 124, 56]. The most important characteristics of piecewise linear functions is that they can be updated efficiently, which is crucial for our workload. We denote our implementation of the function in the data nodes as linearly predictable segments (LP-segments). Figure 5.2 depicts the LP-segments in green, where they are connected to improve range query performance (specifically for high-selectivity range queries).

Due to the nature of a learned model, the function only estimates the data location, where the predicted position may not precisely match the actual location. To control the accuracy of the estimation, learned indexes employ an error threshold to limit the maximum deviation. The error threshold is the maximum distance between the actual and predicted locations. Selecting an appropriate error threshold depends on the data distribution, a challenge for streaming workloads as the data distribution constantly shifts. Therefore, we propose to dynamically adjust the error threshold based on the current search cost using an auto-tuning algorithm (details in Section 5.3.6). Another benefit of this method is that it makes FLIRT parameter-free, unlike other error-bounded learned indexes [78, 72].

We also introduce two concurrent variants of FLIRT: ParalleLPartitionedFlirt (PPFlirt) and ParallelSharedFlirt (PSFlirt), designed to improve search with asynchronous query processing. PPFlirt minimizes the amount of data in each thread with data partitioning by storing a local FLIRT in each thread. Queries are executed independently in each search thread, while an update thread handles enqueues and dequeues. On the other hand, PSFlirt stores a single, unified FLIRT without partitions. Threads can freely access any part of the index. Therefore, the update thread needs to lock the front and back of the index during enqueues and dequeues. Both variants only scale up the search thread since updates only occur at the front and back of the index (details in Section 5.4).

Lastly, we provide FLIRT's aims with respect to existing indexes. FLIRT aims to achieve (1) faster search times that scale with larger window sizes. (2) Faster update performance for monotonic key updates. PPFlirt and PSFlirt aim to improve the search performance further. The experimental results will be presented in Section 5.5.

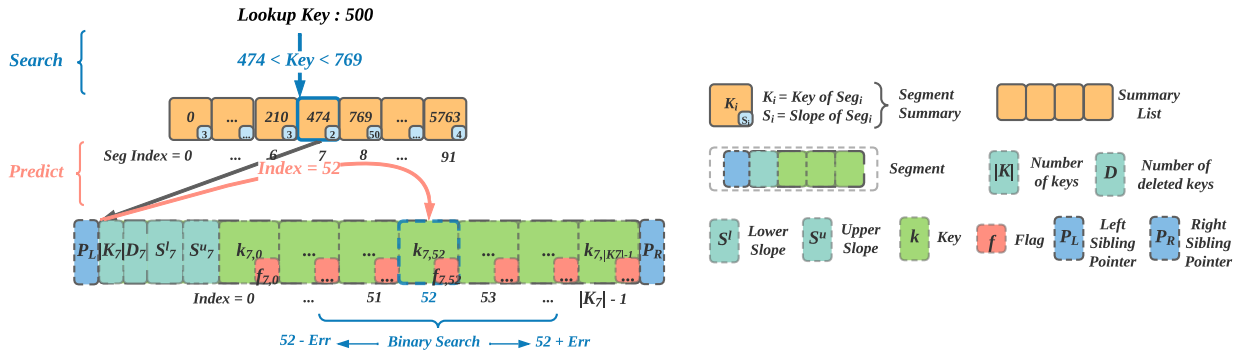


Figure 5.3: Structure of FLIRT illustrating the workings of a search operation.

Table 5.1: Notation table for Chapter 5

Notation	Definition
Err	Approximation error threshold
W	Window size
N	Number of segments
N_{thd}	Number of threads
θ	Degree of skewness
key	Lookup key
seg_i	Segment i
S_i	Slope of segment i w.r.t. $k_{i,0}$
$S_i(key)$	Slope of segment i w.r.t $k_{i,0}$ after inserting key
S_i^u	Upper slope of segment i with respect to $k_{i,0}$
S_i^l	Lower slope of segment i with respect to $k_{i,0}$
D_i	Number of deleted keys in segment i
K_i	Keys stored in segment i
$\|K_i\|$	Number of keys stored in segment i
$k_{i,j}$	Key stored in segment i at index j
$f_{i,j}$	Flag of key stored in segment i at index j
$pos_i(key)$	Predicted position of key in segment i

5.3.2 Index Structure

Table 5.1 presents the notations used for the rest of this chapter. The structure of FLIRT is detailed in Figure 5.3. Starting from the top structure, a circular array-based queue, denoted SummaryList, manages the LP-Segments at the data level. We opt for circular queues for their advantages in terms of memory efficiency compared to linear queues. For similar reasons, we implemented the queue using arrays to reduce memory access compared to linked lists. Each element in the SummaryList stores the starting key and slope of the corresponding LP-Segment to facilitate the prediction of positions without direct segment access. This allows us to directly transverse to the memory location around the prediction position rather than accessing the front of the LP-Segment to find the slope and starting key.

Each LP segment contains a set of metadata, the keys covered by the regression model, and its payloads. The meta-data contains an upper and lower slope that bounds the error to be less than the error threshold. Initially, when there is a single key in the segment, these bounds are set to infinity (upper) and zero (lower). As more keys are enqueued, the upper slope decreases and the lower slope increases, effectively narrowing down the error range [78]. The slope used for prediction (and stored in the Segment Summary) is determined as the midpoint of the slope bound.

To facilitate dequeues, each key has a deletion indicator/flag. This allows keys to be "removed" without repositioning the remaining keys, which avoids retraining the model. This process is known as pseudo-delete. The slots are freed once all the keys in a segment have expired.

5.3.3 Search

The search process starts with the root and identifies the segment that contains the lookup key key in the SummaryList. Formally, each segment seg_i includes a range $k_{i,0} \leq K_i < k_{i+1,0}$, and we search for the last segment whose $k_{i,0} \leq key$. Figure 5.3 depicts an example where $key = 500$ and finds the segment with $k_{i,0} = 474$. We opt for a linear search for array sizes smaller than 256 bytes [2] and use binary search for larger sizes. For binary search, the match condition uses inequality checks: if $K[mid] \leq key < K[mid + 1]$, then key is in seg_{mid}.

Once the corresponding LP-Segment is found, we predict the position using the linear regression model $S_i \times (key - k_{i,0})$, which corresponds to 52 in our example above. Our model guarantees the error between the predicted and actual positions to be within Err. We then apply a binary search between $[pos_i(key) - Err, pos_i(key) + Err]$ to pinpoint the lookup key. Note that this is a lower bound search that finds the smallest key in seg_i that is greater than or equal to key*. Lastly, we check if the two keys match before checking the deletion flag $f_{i,j}$ to verify its validity.

*This search corresponds to the `lower_bound()` function in C++ standard library. It is also known as rank in [72].

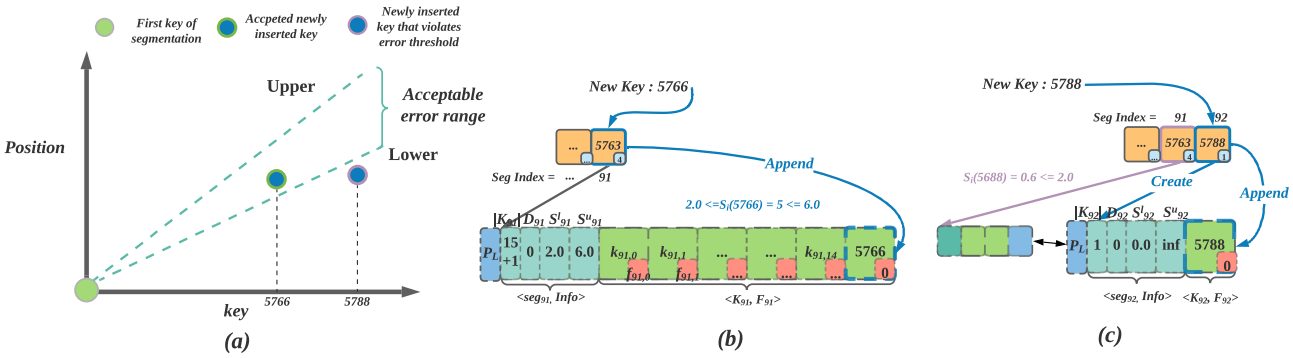


Figure 5.4: Example of an enqueue. (a) Segmentation algorithm (b) Insertion by appending. (c) Insertion by splitting.

5.3.4 Enqueue

Since we must ensure the maximum error in the segment is always less than Err, insertions may result in two scenarios: (1) the maximum post-insertion error remains below Err, or (2) the error exceeds Err. We can check this condition with the upper and lower slopes. Figure 5.4a illustrates the process. Initially, we start with the upper and lower slopes of infinity and zero. Then we incrementally update the slopes with new enqueues using the following equation:

$$S_i^u = \min\left(S_i^u, \frac{|K_i| + Err}{key - k_{i,0}}\right) \tag{5.1}$$

$$S_i^l = \max\left(S_i^l, \frac{|K_i| - Err}{key - k_{i,0}}\right) \tag{5.2}$$

The bound between the two slopes shrinks with incoming keys. The rate of convergence depends on the data distribution. For example, a near linear distribution will cause minimal shrinking, while noisier distributions lead to quicker contractions. If the points fall within the bounds, we get the first scenario, which we show in Figure 5.4b. The key is appended to the segments at an amortized cost of $O(1)$ (we implement the segment as a dynamically allocated array). The slope of the segment is then updated with:

$$S_i(key) = \frac{|K_i|}{key - k_{i,0}} \tag{5.3}$$

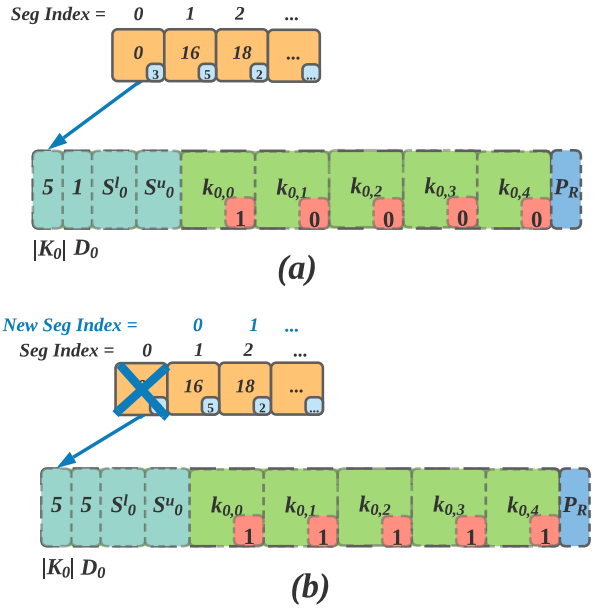


Figure 5.5: Example of a dequeue. (a) Deleting a record in a non-empty segment. (b) Deleting an empty segment.

However, if the new key falls outside the bound, as depicted in Figure 5.4c, we split the segment to prevent violations of the maximum error threshold. Instead of retraining keys in the current segment, the new key forms a starting key of a new segment, which is appended to the SummaryList.

5.3.5 Dequeue

Similar to insertion, there are two cases for dequeuing a record from FLIRT: (1) The segment retains some keys post-deletion. (2) The segment becomes empty. We use a deletion counter to keep track of the number of deletes, which also acts as the deletion index for future dequeues. A segment is empty if the counter equals its total size $|K_0|$ since dequeue can only occur to the first segment.

Therefore, for the first case depicted in Figure 5.5a, if the counter is less than $|K_0|$, we flag $f_{0,0}$ and increment the counter to 1. Once all five keys are deleted in seg_0, we erase the entire segment from SummaryList (shown in Figure 5.5b). The next segment seg_1 (right sibling of the deleted segment) then takes its place.

The rationale for the deletion flag is to mitigate the necessity to shift keys with each deletion. Not only does this reduce the latency from the shift itself, but it also prevents retraining the regression model as the position of keys changes. However, this approach adds an additional check when searching, which we show incurs minimal cost in Section 5.5.

5.3.6 Auto-tune

The search efficiency of FLIRT is closely tied to the choice of the error threshold Err, which determines the binary search cost in each segment and affects the overall segmentation (specifically, the segmentation is determined by the Err and W). Generally speaking, for the same data distribution and W, a higher value of Err allows for more keys per segment, thus decreasing the number of segments. In an extreme case, all records can be encapsulated into a single segment. The segment essentially acts as an array, which results in poor search performance. Conversely, a lower Err has the opposite effect, leading to finer segmentation. At the other extreme, each segment holds two keys (the lowest number of keys to form a linear regression), resulting in a queue and negating any advantages of using a regression model.

While it is possible to pinpoint an "optimal" error threshold for a static dataset through a grid search over Err, the evolving nature of our workload makes it impossible to find a one-size-fits-all solution. Therefore, we propose to use an adaptive Err based on a cost model that aims to balance the search cost at the root (queue) and data (segment) level. We formulate the cost model as follows:

$$C = \underbrace{\log(N)}_{\text{SummaryList Search}} + \underbrace{\log(Err)}_{\text{Segment Search}} \tag{5.4}$$

We use a dynamic gradient descent approach to adjust the Err that minimizes the cost C. Initially, we assume the error threshold to be 256[†] and bulk-loads the keys up to the predefined

[†]This value is determined from empirical evidence from Figure 5.10 and is not the best default value for all distributions. One can run a grid search based on the initial W to find the best initial value. However, we do not see much point since the distribution changes quickly, which also changes the best Err.

window size W. The initial cost of the index is recorded as $C_{initial} = \log(N_{initial}) + \log(256)$, where $N_{initial}$ represents the segment count after bulk-load. Post bulk-loading, we double the error threshold $Err_{new} = 2Err$ and use this value for subsequent enqueues. A tune rate, by default 10% of W, dictates the frequency of cost estimations. Once the number of updates reaches this rate, we estimate the current cost with $C_{current} = \log(N_{new}/tune\,rate) + \log(Err_{new})$, with N_{new} denoting the count of newly inserted segments. Should $C_{current}$ fall below $C_{initial}$, we continue to double the error; otherwise, we reverse the direction and reduce the error by half. Finally, we set $C_{initial} = C_{current}$ and continue the iterative process.

However, similar to the challenge of oversized steps in gradient descent, we may overlook the optimum with static adjustments. Therefore, we employ a step decay strategy by halving the error adjustment if the error threshold oscillates between two values five times. Our dynamic gradient descent differs from the standard gradient descent by increasing error adjustments to accommodate shifting distributions. Specifically, if the error threshold consistently tends to one direction, we double the error adjustment.

5.4 Parallelizing FLIRT

This section introduces the two concurrent variants of FLIRT, ParalleLPartitionedFlirt (PPFlirt) and ParallelSharedFlirt (PSFlirt), that leverage multithreading to accelerate the search. Both variants can have an arbitrary number of search threads but are limited to one update thread. Only one update is necessary due to the sequential nature of the enqueues always occurring at the back, while dequeues only occur at the front. More update threads lead to race conditions increasing the amount of synchronization. The two variants are designed for different workloads: PPFlirt is designed for uniform query workloads, and PSFlirt handles skewed query workloads more efficiently. We now present PPFlirt in detail before introducing PSFlirt.

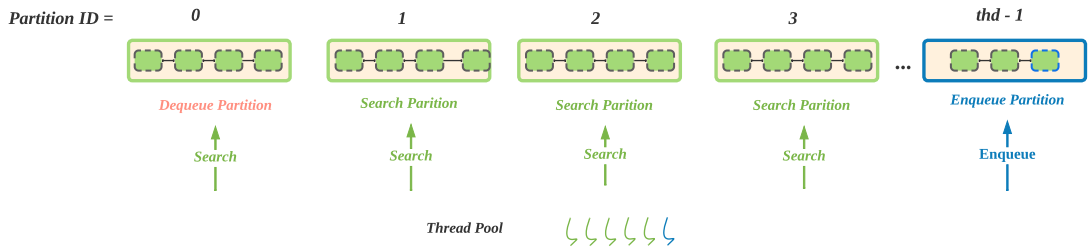


Figure 5.6: Structure of Parallel Partitioned Flirt (PPFlirt), where a thread manages its partition. The enqueue thread (initially empty) is colored in blue.

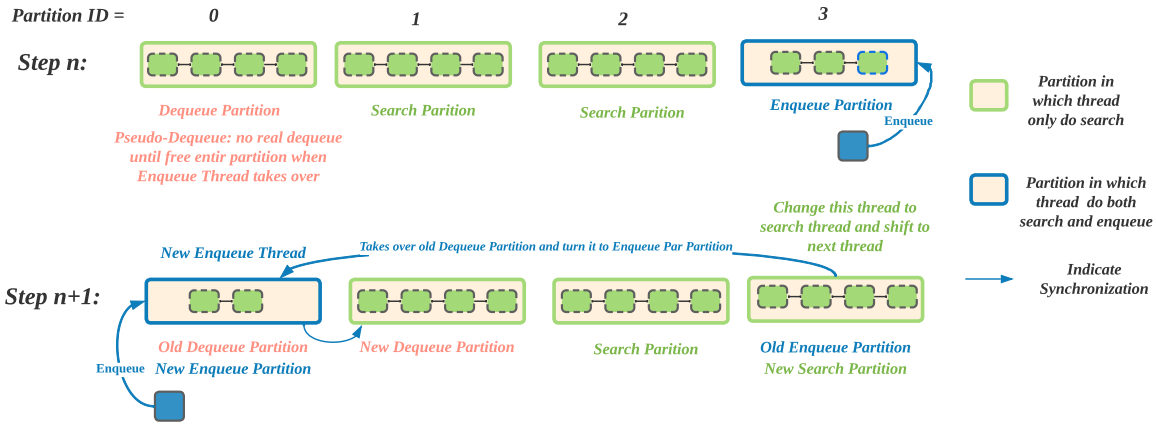


Figure 5.7: Enqueue and Dequeue in PPFlirt.

5.4.1 Parallel Partitioned Flirt (PPFlirt)

The structure of PPlirt is shown in Figure 5.6, where each thread maintains a disjoint partition with a local FLIRT. Each partition has a capacity of $\frac{W}{(thd-1)}$. The -1 ensures one partition is initially reserved for incoming data. The design goal of this structure is to reduce the data volume processed by each thread, thereby reducing the maximum search space (if the model predicts all keys to one location) from N to $\frac{W}{(thd-1)}$.

During bulk load, we divide the partitions amongst the search threads according to ascending key ranges with thread IDs (e.g., larger thread IDs have larger key ranges). Each thread constructs a localized FLIRT and stores the start and end of their key range. The first thread is designated the dequeue thread. The last partition (empty) is assigned to the update thread for enqueuing.

Enqueue & Dequeue: We can exploit the important characteristic of a fixed sliding window, where one enqueue is coupled with a dequeue. We employ a "pseudo" delete strategy to avoid synchronizing these updates by marking keys for removal (with a thread-safe counter) rather than modifying the data in the dequeue partition. Since each partition is independent, the enqueue index will match the dequeue index. Specifically, when we enqueue a key at index 0 in the enqueue partition, we also delete a key at index 0 in the dequeue partition. Therefore, we only need to maintain the index with a thread-safe counter.

For each enqueue and dequeue pair (shown as *step n* in Figure 5.7), the update thread inserts the record and increments the update counter. It then performs a search within its range post-insertion. The search thread responsible for the dequeue partition uses a modified search operation that checks if the position of the key is greater than the counter.

Once the enqueue partition is full, we assign it to a search thread. The previous dequeue partition becomes the new enqueue partition. The transition is depicted as *step n+1* in Figure 5.7, where the dequeue thread (thread 0) is emptied and becomes the new enqueue thread. The dequeue thread always comes after the enqueue thread (in a circular traversal order). Hence, thread 1 becomes the new dequeue thread. The update counter is reset during this process. This transition is carried out solely on the new enqueue thread, which means no synchronization is needed in other threads.

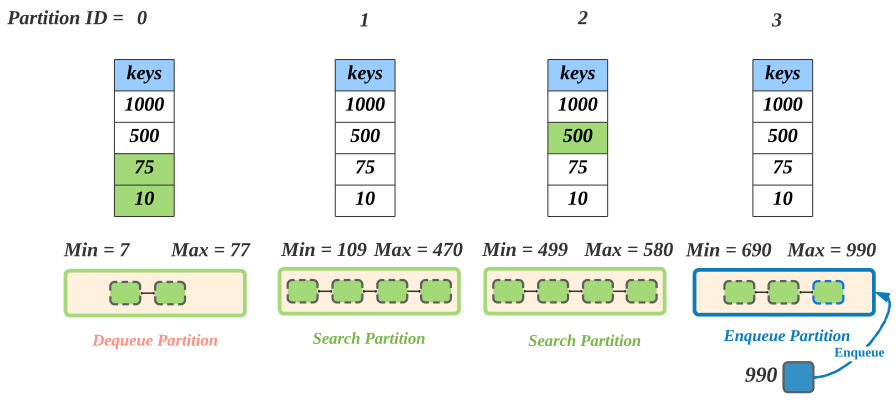


Figure 5.8: Asynchronous search in PPFlirt.

Search: Search is carried out asynchronously in the threads to maximize throughput. The exact execution order is stochastic and depends on the scheduler. However, we can control how the query is dispatched. Traditionally, a range dispatcher is used in a main thread to assign queries to corresponding partitions. The disadvantage of range dispatchers is that they are sequential and require the range of each partition to be synchronized. Therefore, it may cause a bottleneck.

We propose a new search queue dispatcher shown in Figure 5.8. Each thread stores a search queue that asynchronously reads from a queue of queries (which are not dispatched by a main thread). The search queue sees a "view" of the queries to allow threads to address queries independently. A query is erased from the query queue once all threads progress past the view of the query, similar to a sliding window. We ensure a query is carried out only once (for a lookup) by verifying if a query falls within the partition's range (cost of $O(1)$) before any searching. Partitions outside of the query range move on to the next query. The result will be outputted from one thread. We compare the search queue and range dispatcher in Section 5.5.10. Note that both dispatchers are shown not to cause any bottlenecks.

Lastly, we need to take special considerations for range queries as they can intersect multiple partitions. Each thread will check if the lower and upper bounds fall inside its key range. Therefore, multiple partitions can output results for the same query and will be aggregated asynchronously. Hence, the result will be unsorted. We would like to mention a special case where the query range covers the entire partition. In this case, we traverse the entire index without searching.

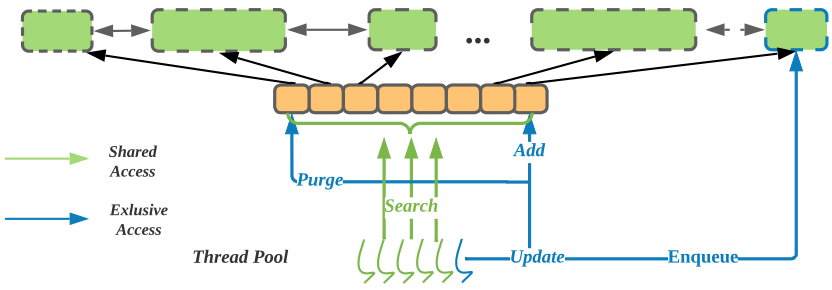


Figure 5.9: Structure of Parallel Shared Flirt (PSFlirt), where all threads share a common "global" index. The update thread (in blue) only accesses the first and last segments.

5.4.2 Parallel Shared Flirt (PSFlirt)

PSFlirt, shown in Figure 5.9, has an opposite design philosophy where threads freely access a unified index. The aim is to distribute the query workload evenly across threads. The gain in throughput is simply from adding more search threads rather than reducing the search space in PPFlirt. Therefore, for a uniformly distributed query workload, PPFlirt will outperform PSFlirt. However, this is not the case for a skewed query workload, where certain data ranges receive a higher concentration of queries. In this case, PSFlirt can outperform PPFlirt by distributing the workload to prevent overburdening individual threads.

Enqueue & Dequeue: Like its counterpart, PPFlirt, PSFlirt also employs a "pseudo" delete to minimize synchronization. The implementation is slightly different since PSFlirt has only one index. The number of segments depends on the data distribution and the error threshold, resulting in asynchronous positions during enqueuing and dequeuing. Therefore, we keep track of the first key in each segment with a sequential ID. The ID indicates the key's position relative to the entire window size and increases with new records. For example, given the window size of 100, the last key will have an ID of 99. The following inserted records are marked as 100, 101, etc.

After enqueueing the data to the last segment, we increment the update counter (which starts at 0). The counter now refers to the sequential ID of the newly deleted key. Therefore, we erase the first segment once the counter equals the first key of the second segment. Since the counter can grow indefinitely, there is a risk of exceeding numerical limits. We avoid this by resetting the sequential ID of the keys and the counter (which requires us to scan through the SummaryList).

Since threads can freely access the index, there is potential for race conditions. For synchronization purposes, PSFlirt implements two levels of reader-writer locks: one at the segment level for insertions without splits and another at the queue level when segments modify the queue structure. The first lock only applies to readers who access the last segment during insertion. The second lock pauses all readers when the first and last segment is erased or split.

Search: Search is conducted by individual threads, each maintaining a queue of lookup keys. Since all threads can access the first segment (where the dequeue occurs), the search algorithm must check the validity of the records by comparing the key's sequential ID with the update counter. For a key at index 10 of seg_i, the sequential ID is found by adding the ID of the first key $k_{i,0}$ to its index in the segment ($ID(k_{i,0}) + 10$).

5.5 Experimental Evaluation

The experimental evaluation examines multiple aspects of the indexes to paint a complete picture. We first analyze how the error threshold affects the performance of FLIRT and validate the capabilities of the auto-tuner. We then compare FLIRT against traditional and learned indexes across workloads and query parameters. To summarize:

- The auto-tuner can dynamically adjust the error threshold for most distributions.
- Some comparative values (in terms of overall latency):
  - *Traditional Indexes:* up to a 6.9× and 8.0× speed-up against a B+Tree and a circular queue, respectively.
  - *Updatable Learned Indexes:* up to a 108.5× and 5.0× speed-up over ALEX and the dynamic PGM-index, respectively.
  - *Streaming Index:* up to a 10^5× speed-up over LIST.
- PPFlirt and PSFlirt improve the search throughput by up to 32.0× and 4.1× over FLIRT, respectively.

5.5.1 Experimental Setup

Dataset & Setup: All indexes and experiments are developed in C++ and tested using the `gcc` compiler on an Intel E5-2680 v3@2.50GHz machine running Centos 8. We use the datasets

from a widely recognized learned index benchmark: the SOSD benchmark [108], comprising of three real-world datasets (*books*, *fb*, *osm*) and four synthetic ones (*lognormal*, *normal*, *uniform dense* (*udense*), and *uniform sparse* (*usparse*)). Streaming datasets (YSB [42] and Stock [214]) are not used as they are used to benchmark how the streaming system schedule the data (e.g., varying arrival and departure rates). Our focus is to highlight performance for different data distributions, which is more crucial to learned indexes.

Baseline: The baselines are introduced in Sections 2.3.3 and 2.4.4. Specifically, the implementation details for the indexes are STX implementation of B+Tree [29], array-based circular queue, PGM-index from [92], and ALEX from [134]. We also include LIST as a streaming index (details in Section 5.2).

We did not include XIndex and FINEdex as they were not released at the time of this project, we will leave the comparison for Chapter 6. Learned LSM-trees, Bourbon [46] and REMIX [218] are not included as they do not describe deletions in their works.

Default Workload: The experiments are simulated with a tuple-based sliding window with a constant window size. The default window size is 100 million keys. The default iteration involves an update (enqueue + dequeue) followed by a lookup task[‡] and is repeated 1 million times. This mimics the progression of a sliding window. The default tuning rate for the auto-tuner is 0.1% of the window size.

Skewed Concurrent Workload: We simulate a skewed workload with a Zipfan distribution with skewness level θ ranging from 0.1 to 0.9. The distribution of queries Q in each thread i is determined as follows:

$$|Q_i| = \frac{|Q|}{i^\theta \times \sum_{j=1}^{N_{thd}} \frac{1}{j^\theta}} \tag{5.5}$$

For PPFlirt, more queries are distributed to the initial threads as θ increases. $|Q_i|$ determines the number of queries in thread i. For PSFlirt, $|Q_i|$ determines the proportion of queries sampled

[‡]same result as a `lower_bound()` in C++.

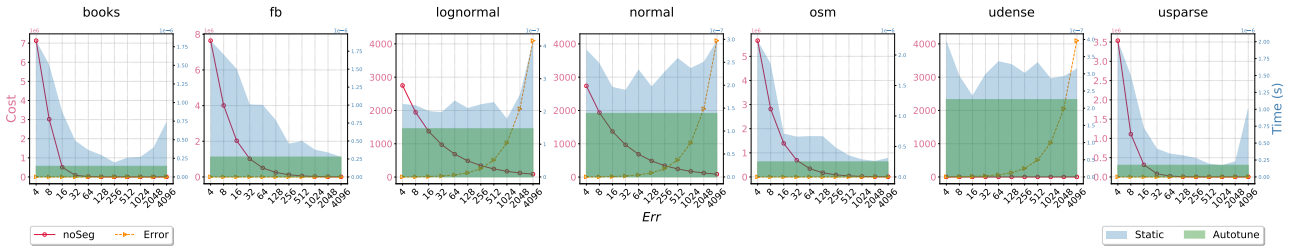


Figure 5.10: Comparison between auto-tuned and static error thresholds.

from smaller keys and increases with θ. The queries Q are then evenly allocated amongst the search threads. This workload only consists of searches and does not include updates.

Performance Metrics: We measure latency via the C++ standard `chrono` library as the average time per operation. Throughput is used for the concurrent evaluations.

5.5.2 Evaluation of the auto-tuner

We mentioned how FLIRT's efficiency is closely linked to the choice of the error threshold in Section 5.3.6. Our first goal is to identify the presence of a minimum or close-to-minimum error for different distributions to justify using a cost model. We show the results in Figure 5.10 with red and orange lines aligned with the left y-axis. The root level search cost is shown in red ($\log(N)$) and the segment search cost in orange ($\log(Err)$) across an error range from 4 to 4096. Cost intersections can be clearly seen for *lognormal* and *normal* datasets. For the rest of the dataset, the intersection point is barely visible due to a large number of segments. The intersections are more apparent for reduced window sizes.

Subsequently, we assess if the auto-tuner can capture the minimal Err with the shaded regions in Figure 5.10, which aligns with the right y-axis. The blue shaded area shows the latency for different values of Err, while the green shaded area shows the latency when auto-tuning. When auto-tuning, the Err is adjusted automatically and is independent of the x-axis values. The comparison yields two findings: first, the shape of the green region roughly coincides with the minimum error (intersection between the red and orange line). Second, the auto-tuner is able to find the minimum error as the blue-shaded region is a subset of the green-shaded region.

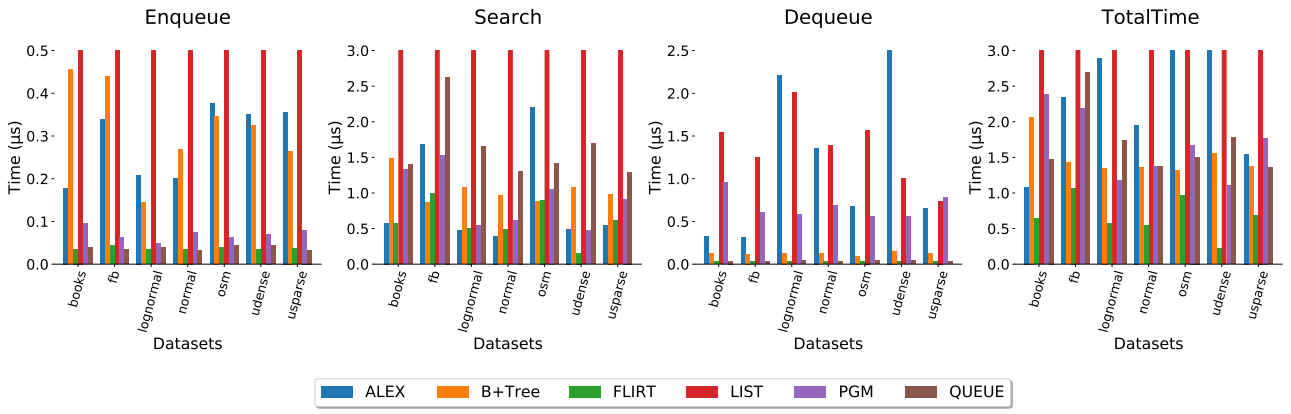


Figure 5.11: Comparative analysis of indexes against FLIRT.

5.5.3 Comparison with the baselines

The B+tree and PGM-index both have parameters. We grid search the branching factor of the B+tree (from 4 to 4096) that results in the fastest execution time for each dataset. The PGM-index underwent a similar optimization for its error parameter. The results are shown in Figure 5.11, where the x-axis shows the different datasets. The y-axis indicates the latency for each operation (enqueue, dequeue, search, and total = enqueue + dequeue + search).

FLIRT demonstrates superior performance compared to the baselines. One advantage of FLIRT is its compact size; the index takes up 0.04% of the window size, significantly smaller than the baselines, which occupy between 5.5% to 14% of the window size. Diving deeper into how FLIRT performs for updates and searches, we see that the search performance dominates the update performance from the latency of the search figure. Additionally, the shape of the search figure predominantly influences the total time. It is also shown to vary with data distribution, suffering mostly in hard-to-learn distributions such as *osm* and *fb*. On the contrary, update performance (enqueue and dequeue) is consistently fast across datasets (on average 10× faster than the search time).

Regarding the baselines, the queue excels in terms of update performance and suffers during the search. This is expected as queues are naturally designed for enqueues and dequeues (where the name of these two operations originates from). The search is simply a binary search over the window size.

The B+tree's performance falls under the search-optimized index expectation, where its update performance is second worst due to the internal balancing mechanisms (split and merge) used to ensure the search performance in each node. These mechanisms do ensure the B+tree performs quite well when searching.

Surprisingly, LIST, the only stream index, suffers considerably due to the implementation of the linked list. The inefficiency is largely due to the cache inefficiency of the doubly linked list in C++, generating a lot of memory accesses, especially when the index size is large (we show the performance worsens with larger window sizes in Figure 5.13). A linked list is also shown to perform horribly during search operations as it requires a lot of memory access and does not support binary search.

In terms of learned indexes, the overall best-performing index is the PGM-index. Its LSM-tree style structure performs well for updates, as it essentially generates a large buffer that can accommodate all enqueue. The main overhead comes from the retraining during the merge step. The search operation benefits from the piecewise linear approximation for a lookup. However, we shall show later that the advantage disappears during range queries.

ALEX is shown to struggle with the sliding window workload. The design of ALEX is optimized for random insertions (we shall see it performs much better in Chapter 6). Regarding enqueues, ALEX optimizes the append-only case by expanding the root node to the right before expanding the last data node rightwards without model re-insertion. However, expanding the root node, potentially generating a new root node, tends to push all existing data to the leftmost child. Therefore, with repeated expansion of the root node, most of the data are stored in a nested hierarchy in the leftmost branch. This generates a highly unbalanced tree, especially for hard-to-learn distributions such as *osm*. Not only does this impact the enqueue performance, but the search performance decreases as we traverse down the leftmost branch. In terms of dequeue, although there are no optimizations, ALEX shows particular weakness when the data exhibited uniform micro-distributions (*lognormal*, *normal*, *udense*).

To summarize, FLIRT has the efficiency of queues for updates while "learning" to search like a learned index.

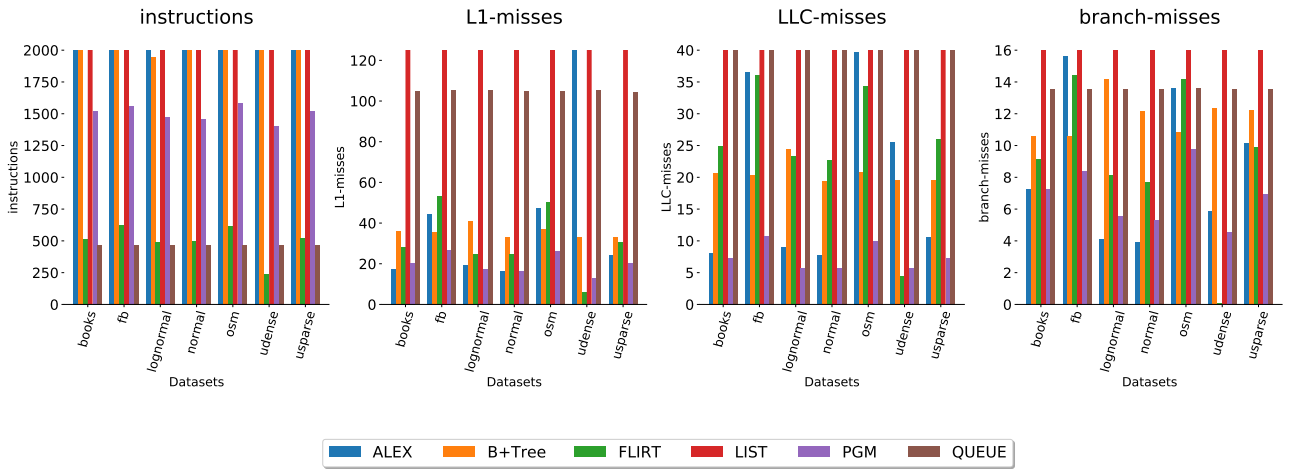


Figure 5.12: Processor level micro-analysis.

5.5.4 CPU-level Micro-Analysis

We employ `PerfEvent`[§] to analyze the processor-level performance of each indexing method. Specifically, we show the L1-cache misses, last-level cache misses, branch misses, and the total number of executed instructions in Figure 5.12.

FLIRT demonstrates a low instruction count, comparable to that of queues, and shares similar L1 cache misses with other learned indexes. Interestingly, the last-level cache misses are much higher for FLIRT, which is closer to indexes with linear structures (queue and LIST). This suggests that the last-level cache miss may be reduced by using a hierarchical tree. Another finding is that the FLIRT's branch misses appear to correlate to the data distribution's linear predictability, which can potentially be used to deduce the error threshold.

In terms of baselines, linear structures are shown to incur a substantial amount of cache misses, while hierarchical structures tend to achieve more cache locality. Intuitively, this makes sense as a queue basically performs binary search, which accesses different parts of the structure. The linked list has no cache or memory locality during pointer chasing. Learned indexes are shown to be able to further decrease the cache misses (even for linear structures like FLIRT). On the opposite spectrum, the number of instructions is shown to be lower for linear structures compared to hierarchical structures, as they are less complex to traverse.

[§]`https://github.com/viktorleis/perfevent`

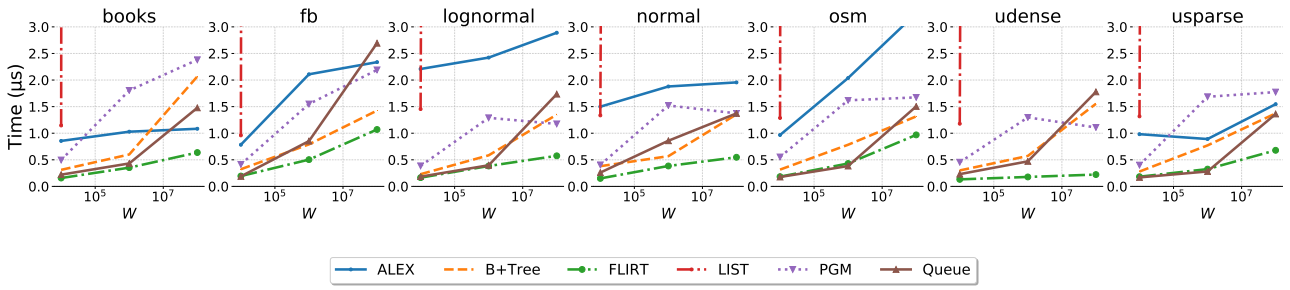


Figure 5.13: Comparative analysis for increasing window sizes.

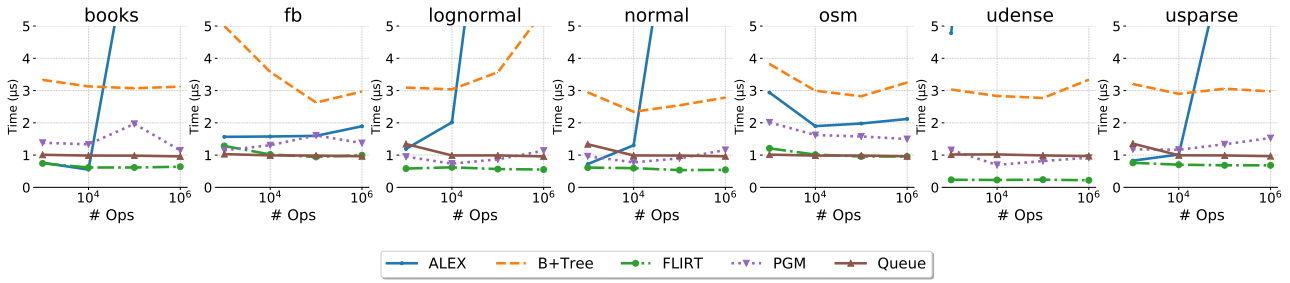


Figure 5.14: Comparative analysis for simulating continuous updates.

5.5.5 Window Size

To evaluate the performance of the indexes at different data sizes, we varied the window size from 10,000 to 100 million. The findings are presented in Figure 5.13, where FLIRT consistently delivered superior performance across all window sizes and datasets. Note that the performance of all indexes tends to converge with smaller window sizes as they start to entirely fit in the cache.

The branching factor and error of the B+tree and the PGM-index are different for each window size, as mentioned in Section 5.5.3. Therefore, we cannot conclude anything about the slope of the lines on their own. However, comparing the B+tree and PGM-index (which are both dominated by the search time shown in Figure 5.11), we can observe that the advantage of the learned component (reducing the search range) becomes more noticeable at larger window sizes. The performance of queues, which is also dominated by the search cost, shows that at 1 million records, cache efficiency decreases when carrying out a binary search. ALEX's inconsistency and poor performance can be attributed to its poor dequeue performance.

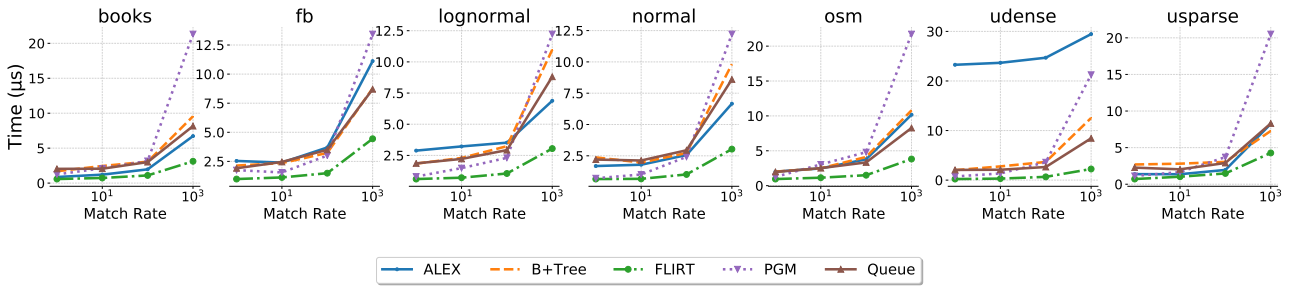


Figure 5.15: Comparative analysis using range queries.

5.5.6 Number of Iterations

To evaluate the indexes' performance under continuous updates, we increase the number of iterations from 1,000 to 1 million with a constant window of 100 million records. This simulates the change in macro-level distribution from a sliding window. Figure 5.14 suggests that shifting macro-level distributions do not heavily affect most indexes. The only exception is ALEX. However, the reason is not because it depends on the macro-level distributions but because the number of enqueues directly determines how unbalanced it becomes. LIST is not shown as the average latency is $2.5 \times 10^4 \mu s$.

5.5.7 Range Query

We then examine range queries by increasing the match rate from 1 to 1000. The match rate indicates the minimum number of records scanned per query. The results are shown in Figure 5.15, we again omit LIST with an average latency of $3.2 \times 10^4 \mu s$.

For linear structures (LIST, queue, FLIRT) and balanced indexes such as the B+tree, the scan range is the actual number of scanned records. Therefore, the slopes for different datasets are similar[¶]. FLIRT is shown to have a different slope for *udense* because there is only one segment. Although ALEX's structure follows that of the B+tree, the gaps designed for random insertion mean that the actual scan range is larger than the match rate. Since it is an unbalanced tree, more traversals are needed.

[¶]Note that this is true for the B+tree; the only reason the slope is different in Figure 5.15 is because we change the fanout.

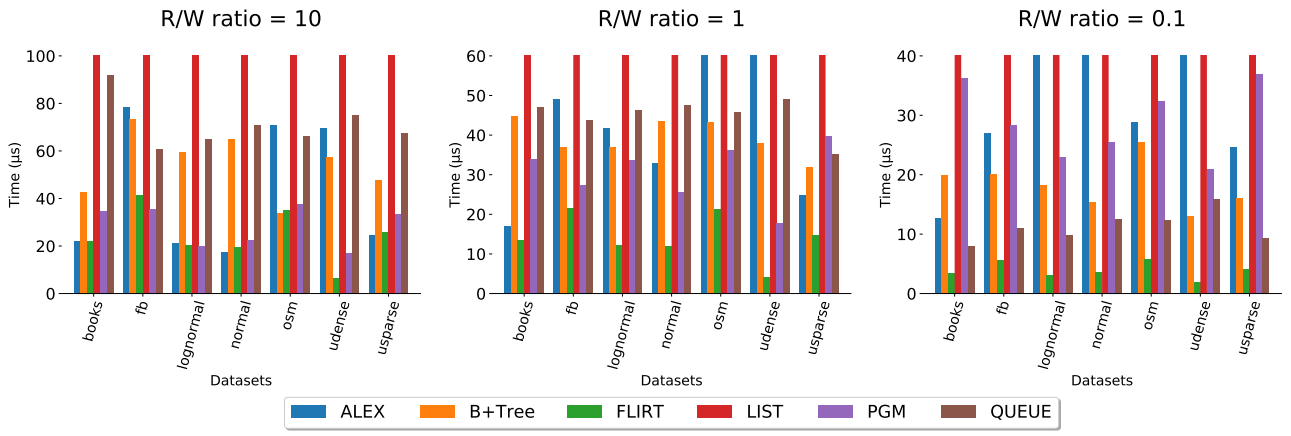


Figure 5.16: Comparative analysis under varied read-to-write ratios.

The disadvantage of the LSM-style structure of the PGM-index becomes more apparent with longer-range scans. The reason is that key ranges can overlap between different levels, which requires us to search and scan multiple levels of the tree. Since the keys are inserted in order in our enqueue and dequeue workload, the disadvantage is not as critical since the smaller keys will always reside in the deeper levels. We will see how the disadvantage worsens with random insertions in Chapter 6.

5.5.8 Read/Write Ratio

Finally, we showcase the performance for different read-to-write ratios from 10:1 to 1:10. Figure 5.16 presents the results which align with the observations in Figure 5.1. Tree-based learned indexes and B+Trees are efficient for read-heavy workloads, while queue-type structures favor write-heavy workloads. FLIRT excels in both workloads, matching the performance of PGM-index and B+tree for read-heavy cases. Its queue-type structure outperforms others in write-heavy tasks.

5.5.9 Parallel FLIRT Performance

For the comparison between PPFlirt and PSFlirt against FLIRT, we set the window size and total number of updates to 50 million (enqueue + dequeue). The number of threads ranges from 4 to 24. Given the concurrent nature of search and updates, we use the number of updates as

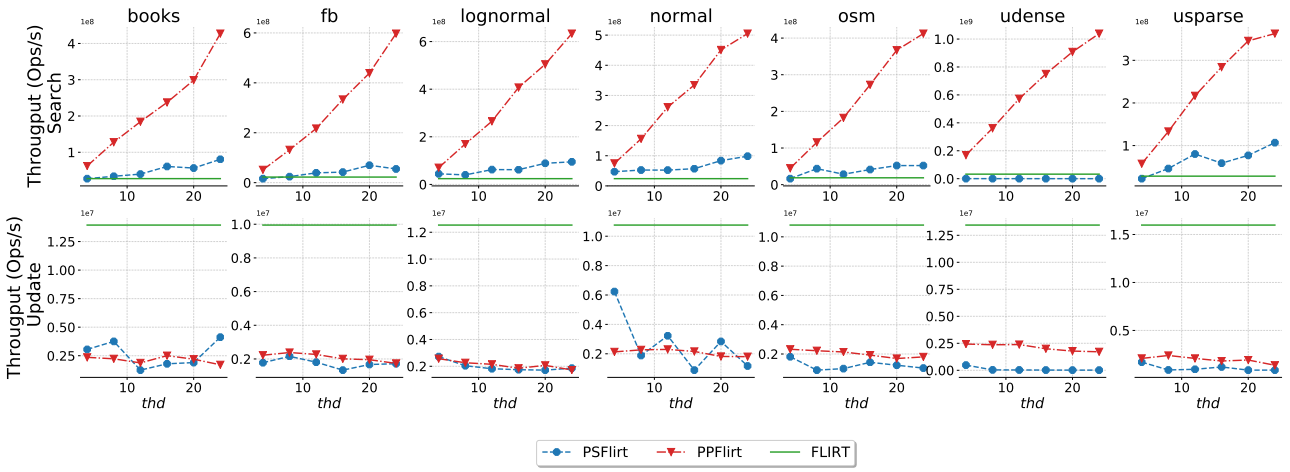


Figure 5.17: Comparison of concurrent variants of FLIRT with single-thread FLIRT (in green) for uniform query workloads.

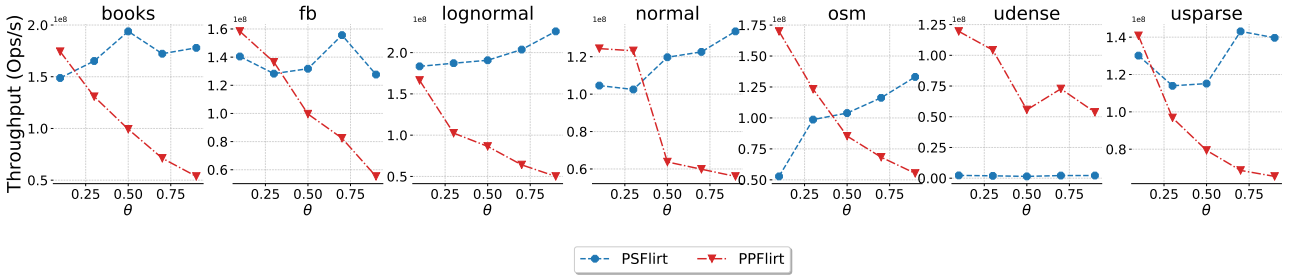


Figure 5.18: Comparison of concurrent variants of FLIRT for skewed query workloads.

the anchor and measure the time to complete 50 million updates. We keep track of the number of searches that occurred during this time. The update throughput is defined as 50 million divided by the execution time. Similarly, the search throughput equals the number of searches divided by the execution time. Note that for PPFlirt, each lookup key is counted once by the thread responsible for processing it.

In terms of search performance, Figure 5.17 shows that PPlirt achieves higher scalability and search throughput relative to PSFlirt. This can be attributed to its lower search cost (roughly $\log(N/N_{thd}) + \log(Err)$) compared to the PSFlirt search cost, which is the same as FLIRT ($\log(N) + \log(Err)$). In terms of scalability, PPFlirt can achieve a 8.0× increase when raising the thread count from 4 to 24. The smaller search space combined with the scalability allows PPFlirt to measure an average speedup of 23.3× over FLIRT. The extreme optimal case of *udense* achieves a super-linear speed up of 32.0×.

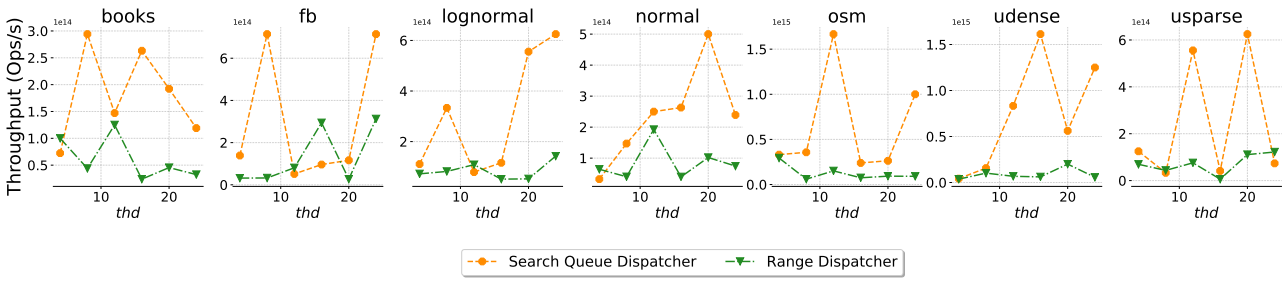


Figure 5.19: Comparative analysis of query dispatchers.

PSFlirt achieves an average of 2.7× increase in search throughput with the same thread variation, resulting in a 2.9× speedup over FLIRT. The overheads from synchronization mechanisms (reader-writer locks) and non-reduced search space are the main performance limitations for PSFlirt. We can observe the overhead from synchronization with the *udense* dataset, where PSFlirt under-performs compared to FLIRT, where both only contain one segment. The highlight of PSFlirt is the skewed workloads shown in Figure 5.18 for a thread count of 12.

In terms of update performance, both PPFlirt and PSFlirt are worse compared to FLIRT. In PPFlirt, the update thread must carry out post-enqueue searches and perform transitions between partitions, while PSFlirt has costly synchronization. To further analyze the synchronization overhead, we can measure the difference in throughput when we have transitions between partitions and when there are no transitions for PPFlirt (mimics the standard deviation of latency) and lock wait times for PSFlirt. Furthermore, we can use CPU usage times (active vs idle time) and thread utilization to profile the performance at the CPU level.

5.5.10 Query Dispatcher Throughput

Lastly, we show the throughput of the query dispatchers introduced in Section 5.4.2 for completeness. Similar to the experiments above, we vary the thread count from 4 to 24 and monitor the duration required to execute 50 million dispatches. The dispatch throughput is 50 million divided by the execution time. For the range dispatcher, a single dispatch thread manages the key ranges in each partition and sends the key to the corresponding partition. In the search queue dispatcher, each thread maintains its key range (Figure 5.8). The dispatch is completed when the thread responsible for the key receives the record.

We show the evaluation in Figure 5.19, where the search queue dispatcher shows an overall higher throughput while being less stable. More importantly, the throughput range is 5 to 6 orders higher than the search and update throughput ranges, eliminating bottleneck concerns.

5.6 Summary

In this chapter, we present FLIRT, a learned index designed for efficiently updating and searching time-ordered keys through a combination of a queue-like structure and piece-wise linear functions. Through an auto-tuning error threshold, it can adjust itself to accommodate data distribution shifts during stream processing. We also present two concurrent variants of FLIRT for increasing the search throughput for uniform and skewed query workloads.

Chapter 6

A Memory-efficient Learned Index for Key-based Streams

6.1 Overview

In this chapter, we extend the problem studied in Chapter 5 into a more general case where keys are not in monotonic order. This general case is known as index-based window processing (IBWP). Since the data stream is still time-ordered, each record now consists of a key and timestamp pair (note that the payload is still present). A more general stream index is key-sorted, while the timestamps control the validity of the data in the stream via inserts and deletes. This allows the index to be looked up by key rather than just timestamps, which accounts for a more general use case. For example, network traffic monitoring systems use sliding windows to store trajectories for real-time monitoring and require querying by IDs rather than timestamps. The challenge for a key-sorted stream index is that it must be able to manage regular timestamp-based updates, which can be randomly inserted into the index (not just at the back) while efficiently searching through keys.

Traditional IBWP indexes incorporate several data structures, such as multiple B-trees, to handle sliding windows, which incurs a significant memory burden. Similarly, learned indexes struggle to achieve compactness for IBWP (where updates are based on non-indexed temporal

dimension), as they need external structures to identify expired keys. Alternatively, learned indexes can use extensive scans, but this neutralizes the advantages of learned indexes and defeats the purpose of using an index in the first place.

In this chapter, our primary objective is to design an updatable index that is significantly more compact than current approaches while benefiting from the gains in search performance from learned indexes. We named it SWIX for the Sliding Window Learned IndeX. SWIX employs a flat structure with only two layers to reduce the index's redundancy by lowering the structure's height compared to hierarchical indexes. Additionally, the flat structure reduces traversal and re-balancing latency during search and updates. These characteristics allow SWIX to support parallelism with lower overheads than hierarchical indexes. Therefore, we propose Parallel SWIX as an extension of SWIX to support parallel reads and writes.

At the core of SWIX, we propose a novel data structure, SWarray, which merges the predictive power of learned indexes (to reduce the search space) and dynamic adjustment features for fast structural modifications during updates. We allow SWarray to expand to store new records and shrink the structure during deletes to ensure compactness. To account for adjustments to the data structure during distribution shifts, SWIX utilizes an online auto-tuning mechanism that controls the degree of expansion and shrinkage with minimal cost. We also designed a bulk-load algorithm to consider the case when the distribution is unknown to the auto-tuner during cold starts. The combined features allow SWIX to be lightweight and fast.

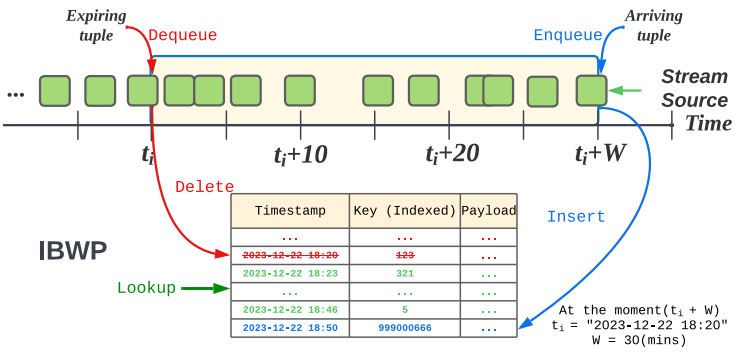


Figure 6.1: A sliding window for index-based window processing.

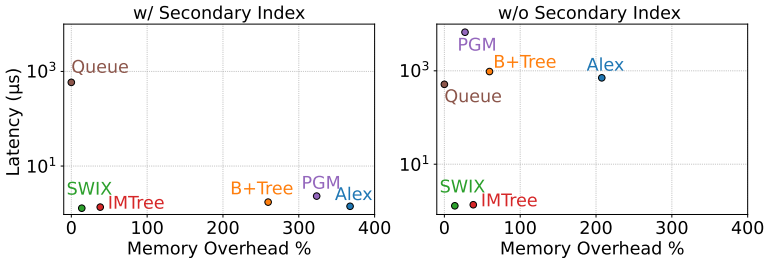


Figure 6.2: Drawbacks of existing indexing in an IBWP workload.

6.2 Index-Based Window Processing (IBWP)

We first formalize the problem of IBWP, which is a generalization of the problem studied in Chapter 5. Figure 6.1 demonstrates how to update a sliding window. New records are enqueued to the window while older records are dequeued. Since each record also has a primary key that is not time-sorted, the corresponding insertions and deletions can occur anywhere in the index.

The sliding window is time-based, where it maintains records from the most recent W time units (e.g., seconds) from the stream. W is known as the window length. Note that the formalization in Chapter 5 is a tuple-based sliding window based on a specific number of records. In contrast, a time-based sliding window does not limit the number of records in a window but rather specifies the duration of each record. Although the implementations are different, both approaches have a notion of data expiry.

6.3 Drawbacks of Existing Work

Current indexes fall short for IBWP workloads either in memory efficiency or total latency, as shown in Figure 6.2. The total latency accounts for enqueue, dequeue, and search with a 1:1 read-write ratio. The two figures show indexes with and without secondary indexes.

The secondary indexes are required for non-streaming indexes (B+tree, PGM-index, and ALEX) to ensure low latency in IBWP workloads. It acts as a timestamp-to-key mapping for finding expired keys. Without secondary indexes, non-streaming indexes need extensive scans to find expired records, since the indexes are only built on keys. However, incorporating a sec-

ondary index is a double-edged sword as it significantly increases the total memory footprint to roughly 3× the data size. This renders both approaches (with or without secondary indexes) sub-optimal for the IBWP workload.

Considering indexes designed for streams, queues are shown to have the smallest size. However, since records in the queue are time-ordered, we can only apply linear search since the keys are unsorted. At this point, the queue is not much of an index. This is the reason why we will omit FLIRT (Chapter 5) in this comparison*. We consider IMTree [162] as our main competitor. The IMTree is originally designed for streaming joins using multiple B+trees. The idea is to use a small dynamic B+tree (CSS-tree) as a buffer for insertions and a larger static B+tree for searching. It tracks both keys and timestamps and expires records when two trees are merged. Despite its effectiveness, it is unable to achieve the same memory efficiency as SWIX.

6.4 The Sliding Window Learned IndeX

6.4.1 Overview

We first present an overview of the design considerations that led to SWIX before describing the components in detail. We start by exploring the idea of using a flat structure to optimize for both space and time. This design decision comes from the insight that hierarchical tree structures, including traditional indexes such as the B+tree and updatable learned indexes [88, 72, 56, 162, 180, 119], have redundancies that are sub-optimal for streaming data. In the case of the B+tree, all internal nodes contain redundant keys to speed up search performance. We can reduce the number of redundancies with a shallower tree by increasing the branching factor. However, this comes at the cost of search performance as the search space in each node increases. The same principle is applicable across tree-based indexes, where there is a clear trade-off between size and search performance. Therefore, we employ a two-level structure that effectively minimizes the number of redundant keys (thereby reducing the index size) without compromising search performance. We argue that carefully designed learned data structures

*SWIX supports the workload of FLIRT which we shall compare in Section 6.6.

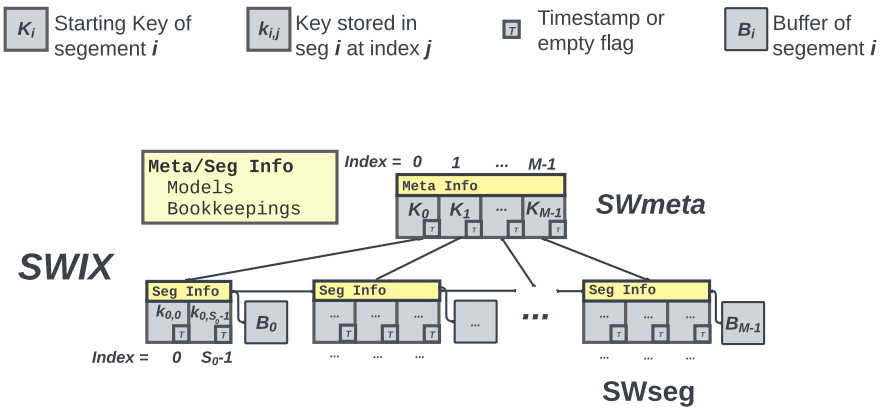


Figure 6.3: Index structure of SWIX

that accommodate shifts in data distribution can achieve this goal. As we shall see in the experimental evaluation (Section 6.6), we cannot guarantee latency improvements from having extra levels in a hierarchical structure, further strengthening our case for a memory-optimized design.

One challenge with having fewer levels is the need for efficient search in more populated nodes. We utilize piecewise linear models to transform the search from a size-dependent problem into an error-based problem. Therefore, we can sustain a high search performance by controlling the error (e.g., reducing error with more models) regardless of each node's size.

Large node sizes also make updates problematic in two ways: (1) more keys need to be reallocated to keep the keys sorted, increasing the update time. (2) Shifting the keys diminishes the model's accuracy, resulting in frequent retrains. To mitigate these problems, we allocate gaps at strategic locations to reduce the amount of keys shifted. ALEX employs a similar tactic of smoothing the data distribution using gaps to prevent consecutive keys from predicting the same position. We believe gaps should be generated and removed dynamically based on data distribution rather than a fixed gap strategy (resulting in wasted space). An auto-tuning mechanism controls the dynamic adjustment by monitoring the index performance with a cost model. Furthermore, we use a node-level buffer to prevent long-distance data shifts and to absorb model accuracy loss. These efforts combine to reduce wasted spaces in the node while minimizing the number of retrains.

Lastly, we introduce parallelism by segmenting SWIX into separate partitions. Parallel SWIX allows for parallel reads and writes and differs from existing concurrent approaches [180, 119]. As shown in Chapter 5, partitioning can effectively generate a smaller workload in each processor. Futhermore, we allow each partition to perform lock-free updates and retrains.

6.4.2 Index Structure

Figure 6.3 present the structure of SWIX. Each node uses a variant of the base data structure SWarray that incorporates a linear prediction model and a dynamically sizeable (shrink & expand) contiguous array. It also supports implicit deletions (using timestamps) without a secondary index. We now show the variants of SWarray for the segment and root level:

SWseg: The segment level nodes are denoted SWsegs and store the records. They are also interconnected via pointers to enhance long-range queries. SWseg is prone to data shifting from updating records. Therefore, it is also adjusted the most. To accommodate insertions, we allocate gaps by modeling the proximity of existing keys with the trained linear model. Specifically, if two consecutive keys have a tighter key range (e.g., 1 and 2), it implies a reduced likelihood of gaps to accommodate other keys. Conversely, a wider key range (e.g., 1 and 10) gives a higher potential for gaps. Note that the auto-tuner determines the number of gaps according to the current data distribution.

Although we can easily estimate the likelihood of gap location for static datasets, it becomes much harder when the data distribution shifts (especially when the concentration of keys in different key ranges shifts). Instead of expanding the gap count, which increases space overhead and gaps in non-concentrated ranges, we use a cache-efficient universal buffer to accommodate insertions in dense areas. Any insertions that lead to long-distance data shifts (indicating a continuous range of non-expired keys) will be inserted into the buffer. The additional benefit of the buffer is its ability to store records without incurring accuracy penalties. Therefore, we also use the buffer to store new keys when the error is at the acceptable maximum. The buffer is implemented as a fixed-sized sorted array.

To accommodate deletes, we flag the expired record in SWarray to prevent shifting. The flagged positions automatically become new gaps for insertion. We control the number of gaps to prevent inefficient space utilization and poor scan performance by compacting the segment (details in Section 6.4.5). Additionally, records are erased from the buffer to maintain compactness.

SWmeta: At the root level, we have a single SWmeta that manages all SWsegs through the starting key of each segment. Since SWsegs buffer the distribution changes from the update, modification to the root node is much rarer, occurring only when one or more segments are retrained. The main goal is to ensure a low search latency while maintaining a small size (since the root node essentially consists of redundant keys). We, therefore, refrain from using a buffer. Gaps are indicated with a bitmap representation for fast traversals[†]. Additionally, we store another bitmap to support collective retraining of SWsegs to minimize fragmentation.

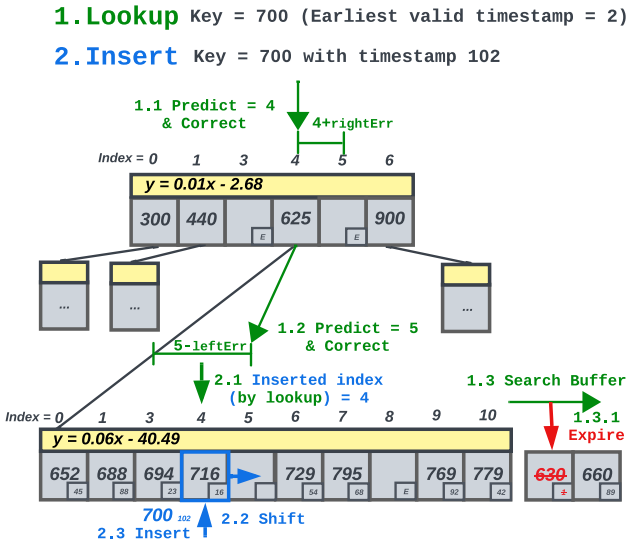


Figure 6.4: Illustration of search, insert, and delete operations using SWIX. The search process is shown in green, where we employ two *predict-correct* steps to find the key 700. We search the buffer since the key is not in the array. The expired data (key 630) is implicitly deleted in this process, shown in red. We then insert the same key at position 4, which causes a shift (shown in blue).

[†]A bitmap compacts an array of Boolean values, with each Boolean represented by a single bit. We implement the bitmap with a C++ vector of 64-bit unsigned integers, which allows 64 Boolean values per integer. The size of the bitmap is `ceil`(number of items/64).

6.4.3 Search

The procedure for finding a key k in SWIX revolves around a two-step *predict-correct* method: first, we predict the position of the key using the model. We then correct the prediction error with a search. We optimize the search algorithm in the correction step with a bounded exponential search. The insight for using exponential search over binary search is that our insertion strategy places new records close to the predicted position. We will elaborate on this in Sections 6.4.4. The search bound provides a worst-case performance for the correction step and allows us to estimate the search cost in the auto-tuner.

We start at the SWmeta to find the segment that holds the key range containing k, illustrated in Figure 6.4. Specifically, we find the last segment i such that the starting key k_i is larger than k. The *predict-correct* procedure is then applied to the corresponding segment. A binary search is conducted within the buffer if the key does not appear in the SWarray. Once the lower bound or rank of k is found, we check if its timestamp is within the time window (updated during insertions). For range queries, we look at the lower limit of the range and scan through records until reaching the upper limit (skipping over any expired records). Neighboring segments are traversed via pointers if the query range covers multiple segments.

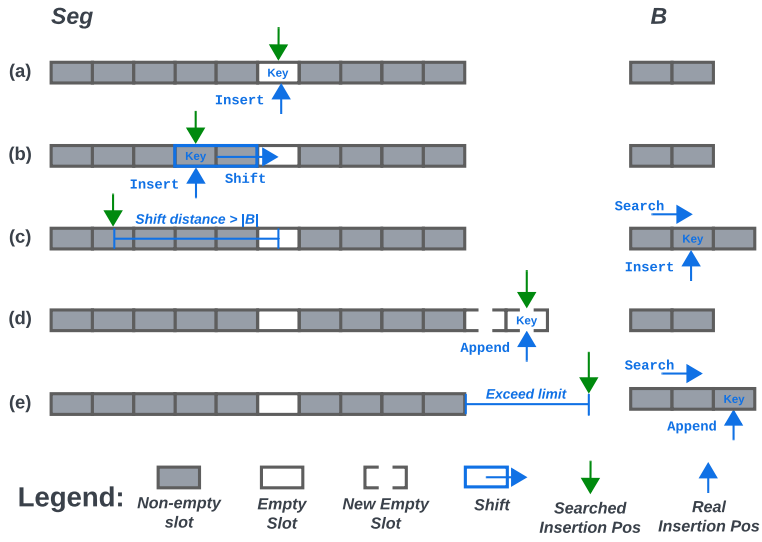


Figure 6.5: Different cases of segment modification during insertion.

6.4.4 Insert

We use a lookup to find the position for the new record to ensure a sorted order. Once the position is found, we have several insertion strategies, depicted in Figure 6.5, depending on the status of the position. If the position is a gap or the existing key has expired, we insert the key (Figure 6.5a). If not, we must relocate some keys by shifting the adjacent keys towards the nearest available gap. This ensures the inserted keys remain in proximity to the predicted position, as depicted in Figure 6.4 & 6.5b. We show a right shift in these two figures as the closest gap is to the right; the shift can occur in both directions. This generates an accuracy penalty, which we account for by incrementing the error bound in the shift direction.

We make two optimizations to reduce the impact of shifts. (1) We limit the accuracy penalty from shifts by capping the number of shifts to 30% of the SWseg size. This value is dynamic for each segment as they have different sizes. Once this cap is reached, all inserts will go into the buffer (Figure 6.5c). (2) We minimize the memory reallocation from shifts by restricting the shift length to the buffer length $|B|$. Since it would be cheaper to insert records into the buffer for any shift length greater than $|B|$, with the added benefit of not having accuracy penalties, this also means that our maximum insertion cost is $|B|$. Once the buffer is full, we flag the segment in SWmeta for retraining.

We also account for out-of-range insertions or appends (the main type of insertions seen in Chapter 5). In this case, we allow extrapolation insertions to minimize prediction errors instead of confining appends to the last element like existing indexes. We illustrate an example in Figure 6.5d, where we extrapolate the position of the key and expand the array to that position. To mitigate the creation of excessive gaps between the last element and the extrapolated position, we restrict the number of gaps to 50% of the initial segment size, denoted as the maximum gap threshold. If the threshold is exceeded, we append the key into the buffer as shown in Figure 6.5e.

Lastly, the window is updated if the record's associated timestamp t is the largest. We update the window by finding the expiry timestamp $t_e = t - W$, where W is the window length.

Algorithm 6 Segmentation Algorithm for Bulk-load

```
1: procedure SPLIT_SECOND_DERIVATIVE(K)
2:     Input: K: collection of keys, where |K| is the number of keys.
3:     Output: Split: collection of index in K to split.
4:     d²(pos)/d(k)² ← second derivative for x = [k_0, k_|K|−1], y = [0, |K| − 1]
5:     while N < N_threshold do
6:         pos = max(d²(pos)/d(k)²)
7:         if pos = 0 then
8:             break
9:         end if
10:        Split.push_back(pos)
11:        d²(pos)/d(k)²[pos] = 0
12:        d²(pos)/d(k)².penalize_neighbors(pos)
13:    end while
14: end procedure
```

6.4.5 Delete

We reduce the deletion cost by employing implicitly expiring records during searches, scans, and inserts. The idea is to amortize the cost of an explicit delete into other operations. The key insight here is that other operations already access the records. Therefore, adding a small check would be cheaper than traversing the index to find a record to delete, not to mention the extra cost of searching a non-indexed dimension (timestamp). Therefore, we expire records when they are accessed in other operations (e.g., during exponential search, scanning, or shifting) by comparing their timestamps with the expiry timestamp t_e. Invalid records are transformed into gaps within SWarray and removed from the buffer. To mitigate excessive gaps from deletion, we oblige to the maximum gap threshold (50% of the initial segment size). Note that if the entire segment expires, we flag it as a gap in SWmeta.

We would like to mention the case for tuple-based sliding windows (seen in Chapter 5). In this case, SWIX treats the sequence order of the records as timestamps. For a n-sized dataset, where n is greater than the time window W (which is technically the window size in this case). If the latest timestamp is n-1, the expiry timestamp t_e is $(n-1)-W$.

Lastly, SWIX also supports explicit delete via a key lookup for cases where users may want to manually remove a record (e.g., invalid entry). Similar to implicit deletes, the target record is transformed into a gap within SWarray or removed from the buffer.

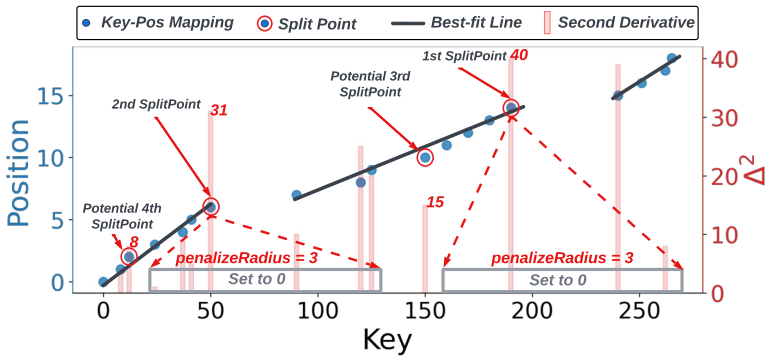


Figure 6.6: Example of second derivative segmentation algorithm for cold starts.

6.4.6 Bulk-load

The objective of the bulk-load algorithm is to organize the data such that the error is minimized during cold starts when the auto-tuner has not yet captured the data distribution. Our algorithm is based on reducing prediction errors of linear models. Hence, the data is split so that each segment has a near-linear distribution while minimizing the number of segments to ensure a lightweight structure. We introduce a second-derivative algorithm employing a top-down splitting approach. The underlying principle is that the second derivative indicates sudden changes in data distribution, which means segmenting these values allows us to reduce the error quickly.

The workings are shown in Algorithm 6 and Figure 6.6. We find the second derivative between the data (shown in red and corresponds to the right y-axis). The algorithm then recursively identifies the peaks as potential split points. To prevent segments from being too short and the concentration of split points at small key ranges, we avoid splitting points that are too close. E.g., in Figure 6.6, the second peak with a second derivative of 39 is neglected since it is next to the first split point. We stop the algorithm after reaching a predetermined number of segments $N_{threshold}$, which we heuristically determined to be 0.01% of the data size n by default. There is also an early termination criterion when the remaining peaks during recursion are uniform. Although further splitting may reduce the error, it is more likely to cause data fragmentation and wasted space. An extreme case would be if all remaining second derivatives are zero, where a single model is sufficient to index the remainder of the data.

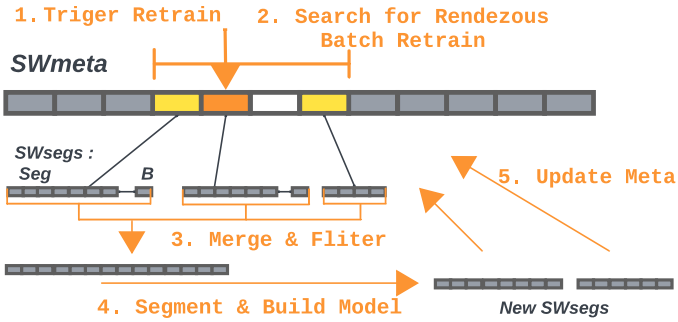


Figure 6.7: The steps for a rendezvous batch retrain of segments.

6.4.7 Retrain

Structural modification from updates leads to accuracy penalties in learned indexes. Retraining aims to correct errors with a snapshot of the current data distribution. In addition to reducing latency, SWIX aims to compact itself using the data distribution by correlating the size to the linearity of the data distribution. Linear distributions will allow smaller, densely packed segments, while more gaps are reserved in the segment for non-linear distributions. Any changes in distribution are captured by the auto-tuner, which determines the size of the segment during retraining. This differs from the predefined bounds used by current approaches, which we argue is space inefficient. We now present how the segments are retrained at the two levels.

Retraining SWseg: We start with the segments as our retraining process is bottom-up. The goal of segment retrains is to correct the errors while preventing data fragmentation (smaller and smaller segments). Having smaller segments increases memory usage and reduces the search performance for two-layer structures (less of a concern for hierarchical structures). Therefore, we employ a technique known as rendezvous batch retraining, where segments wait for neighbors to be retrained together. This technique also allows key ranges from segments to regroup for a lower average error.

We show the steps of the algorithm in Figure 6.7. Upon initiating a retrain (step 1), we identify all adjacent segments requiring retraining (step 2). There may be cases when we find no neighbors. To avoid data fragmentation, we allow the segment to wait for their neighbors and skip retraining. However, if the same segment triggers another retrain, we carry out a mandatory retrain regardless of whether any neighbors are found. Then, the data in the

segments are sort-merged, and expired data are removed (step 3). For a single segment, the buffer is merged with the array. Lastly, we apply a one-pass segmentation algorithm to split the segments (step 4).

The segmentation algorithm is controlled by a *tol* parameter defining each segment's maximum allowable error. The auto-tuner adjusts *tol* based on the current distribution. We use an error-based parameter rather than a size-based one to account for segment-level data distribution, generating three benefits: (1) it reduces the average error by grouping key ranges by error. (2) We can minimize the size while guaranteeing the error. (3) It provides a theoretical guarantee of search cost. For example, if we use split segments by powers of 2 (as seen in existing indexes), we cannot guarantee that each segment has a close to linear distribution as the data distribution is not correlated with data size. If the data turns out to be a continuous sequence of numbers (perfectly linear), we do not need to split the segment at all to achieve a good search performance.

Retraining SWmeta: In step 5, new segments are passed on the root level. SWmeta flags old segments as gaps and inserts new segments. The overall insertion process is the same as the SWseg insertion process outlined in Section 6.4.4. The only difference is that there is no buffer in SWmeta.

Since insertions are rare in SWmeta and primarily involve key replacement, we allow the root node to expand more freely by scaling the linear model if there is insufficient space. The scaling generates more gaps evenly across the node, which also reduces the average shift distance. We prevent excessive gaps by retraining the SWmeta if the gaps exceed 20% of the node size. Retraining SWmeta involves removing all expired segments, which gives us a new node size. We then retrain the linear model and scale the model to include gaps. The number of gaps is 5% of the new node size before scaling.

6.4.8 Analytical Cost Model

In this section, we provide an analytical cost model of SWIX based on the maximum time cost of each operation. The operations include search (C_S), insert (C_I) and retrain (C_R). Delete is not included since it is done implicitly. For completeness, an explicit delete will cost the same as a search.

$$C_{SWIX} \begin{cases} \log(\epsilon_{meta}) + \log(\epsilon_{seg}) & C_S \\ C_S + \alpha|B| & C_I \\ \beta_{seg}((A_i + B_i) + N_{seg}) + \beta_{meta} N_{seg} & C_R \end{cases} \tag{6.1}$$

We start by explaining the search cost, which comprises the error bound at each level. ϵ_{meta} represents the maximum search bound for the exponential search in SWmeta. Similarly, ϵ_{seg} is the error bound for a SWseg, which is equal to *tol* after retrains. The two error bounds exhibit an inverse relationship, assuming the data distribution stays constant. A smaller ϵ_{seg} typically generates more segments, thereby raising ϵ_{meta}. Note that the relationship is not 1:1 and depends on many factors, such as gap availability and distribution.

The insertion cost is the search cost plus any structural modifications required to keep the data sorted. No modification is required if the predicted position is empty. The most expensive modification is from shifting either in the SWarray or the buffer. Therefore, the maximum cost is $|B|$. We use α to represent the proportion of inserted keys that cause shifting in the SWarray or inserted into the buffer.

The retrain cost can be broken down into segment and root costs. The segment retrain cost ($\beta_{seg}((A_i + B) + N_{seg})$) includes the merge cost of the retraining segment and the insertion cost of new segments. $(A_i + B_i)$ is the cost of merging the array A_i and buffer B_i of segment i[‡]. The cost of inserting a new segment into SWmeta at most the total number of segments N_{seg} (shift all records in SWmeta from start to end). β_{seg} is the frequency of segment retrains. It is

[‡]The maximum merge cost if all segments retrains is $\sum_{i=0}^{N_{seg}-1}(A_i + B_i)$.

proportional to the frequency of root retrains β_{meta} as more insertions in SWmeta increase the likelihood of triggering a root retrain. The root retrain cost is N_{seg}.

6.4.9 Auto-tuner

We developed a heuristic cost model to understand how various internal states influence the index performance in terms of *tol*. The cost model allows us to estimate the cost without costly timing mechanisms. Each component within the cost model represents a cost for an operation (e.g., search). More specifically, the cost represents the impact of an internal state on model accuracy. The internal states were discovered with multiple experiments on the relationship between *tol* and the latency of each operation for different datasets and index configurations. We exclude states and operations with no relationships. For example, the insertion cost is determined by the shift distance, which shows no relationship with *tol*. We then select states using an Akaike Information Criterion (AIC) [9] to determine the significance of each state by iteratively incorporating and excluding states. The final states are combined to establish the following cost model:

$$\begin{aligned} C_{tune} = w_{search1} \log(\overline{L}^{meta}_{search} + \overline{L}^{seg}_{search}) + w_{search2}(tol) \\ + w_{scan}(\overline{N}^{pointer}_{traversal}) + w_{retrain}(L_{merge} + L_{retrain}) \end{aligned} \tag{6.2}$$

We first explain what the selected states are before introducing the tuning algorithm. For the search cost, the first component describes the running error captured by the average exponential search length for SWmeta ($\overline{L}^{meta}_{search}$) and SWseg ($\overline{L}^{seg}_{search}$). Note that this component captures the accuracy penalty from shifting. The second component includes the initial error after retraining in the form of *tol*. During range searches, the average number of sibling pointer traversals $\overline{N}^{pointer}_{traversal}$ approximates the scan range. The retrain cost is captured by the merge and retrain lengths, which are in terms of cumulative length to account for the frequency of retrains (which we want to minimize). Weights characterize the importance of each operation. Specifically, $w_{search1}$ & $w_{search2}$ are the weights for search, w_{scan} for scan, and $w_{retrain}$ for retrain.

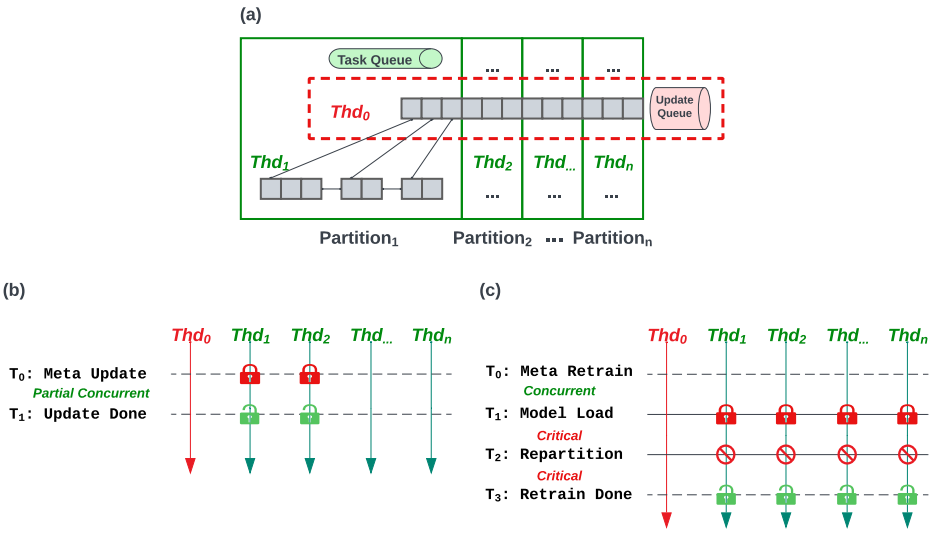


Figure 6.8: (a) Structure of Parallel SWIX. (b) Synchronization during meta update. (c) Synchronization during meta retrains.

The weights are calibrated based on the difference between the model prediction and actual latency.

Since all these states are correlated to *tol* (one of the states is *tol* itself), we can minimize the cost by adjusting *tol* at run-time. The tuning algorithm follows the same idea as the tuning algorithm in Chapter 5, where we perturb *tol* in a direction (increase or decrease) and monitor its impact on cost. We calculate a new cost every time we hit a tuning frequency. If the cost is smaller than the previous cost, we continue in the same direction. Else, we set an opposite course. To avoid fluctuations around two *tol* values, we use step decay to reduce the magnitude of adjustment. Similarly, if the cost continuously heads in one direction (e.g., distribution shifts change the minimum cost), we increase the magnitude of adjustment.

6.5 Parallel SWIX

6.5.1 Index Structure

The structure of Parallel SWIX is presented in Figure 6.8a. We use a meta thread thd_0 to manage the SWmeta, while other worker threads are responsible for a fixed set of segments

through partitioning. The segments are linked within each partition, whereas inter-partition connections are avoided to prevent concurrent access. Although the number of segments is fixed in each partition, the number of data varies as segment sizes are non-uniform. This strategy allows us to ensure load balance by maintaining a similar error between partitions since SWIX's performance is more influenced by error than the volume of data.

All threads are assigned tasks (e.g., search or insert) through a task queue. The task queue for thd_0 receives update requests from worker threads to apply any adjustments that span multiple partitions, such as SWmeta insertion and retrain. On the other hand, the task queue for the worker threads is populated by a query dispatcher.

6.5.2 Parallel Search

Search operations adhere to the *predict-correct* procedure in SWIX, but adjusted for concurrency. When given a query, the dispatcher assigns it to all the worker threads. Each thread gets a view of the SWmeta and predicts the segments responsible for the key. Since the prediction is not accurate, we need to correct the error. To reduce threads not involved in the query, only threads responsible for segments that overlap the error bound will conduct the correction. Once the segment containing the query's key range is found, only the thread that manages that segment will carry out the *predict-correct* process in the segment. Therefore, for a lookup query, only a single thread will return the results. Consequently, the maximum search cost in any worker thread is $O(\log(\min(\epsilon_{meta}, P)))$ for the SWmeta level, and $O(\log \epsilon_{seg})$ for the segment level. ϵ_{meta} and ϵ_{seg} denotes the meta and segment errors respectively, and P denotes the partition size.

For range queries, multiple threads are engaged, as the range can extend several partitions. Let's consider a range [k_l,k_u]. We first predict the segments that contain k_l and k_u. The total error bound of the range would include segments from $predict(k_l) - \epsilon_{meta}$ to $predict(k_u) + \epsilon_{meta}$. For any partition that contains k_l, identified by [$predict(k_l) - \epsilon_{meta}$, $predict(k_l) + \epsilon_{meta}$], we correct for the segment to find the starting segment. Any partitions that contain segments after the starting segment until $predict(k_u) + \epsilon_{meta}$ perform a scan until finding a key greater

than k_u. We avoid correcting for the upper limit since the keys left of the k_u must be scanned, thereby reducing redundant searches.

The final results are aggregated asynchronously based on a thread-safe data structure depending on the query. For example, we can use a thread-safe counter (`C++ atomic`) for count queries. If exact keys need to be returned, we use a pre-allocated pointer array where the size of the array is $number\ of\ threads$ - 1.

6.5.3 Parallel Insert & Delete

The insertion process is a search followed by a segment-level serial insertion (Section 6.4.4). Therefore, the insertion cost is at most $O(\log(\min(\epsilon_{meta}, P)) + \log \epsilon_{seg} + \alpha B)$, where B is the buffer length. Similar to SWIX, deletes are implicitly embedded into other operations (see Section 6.4.5 for details).

6.5.4 Concurrent Retrain

Retraining SWseg: The same rendezvous batch retraining is applied to Parallel SWIX but at a partition level, where the segments are confined to be within the same partition to reduce synchronization. This allows the threads to conduct their retraining asynchronously. Once the retrain completes, the thread flags the old segments as gaps. No race condition will occur here as a segment is managed by a single thread.

However, the insertion of new segments becomes a bit problematic since the closest available gap may be located in a different partition. Therefore, our strategy is to have the thread initially search for gaps within its partition, even though it might not be the closest. If no gaps are found, synchronization must occur, and the thread delegates the task to the meta thread through the task queue. It flags the insertion position and moves to a new task. However, if the new task needs to access the new segment before thd_0 completes the insertion, the thread must stop and wait for the insertion to finish.

When the meta thread receives an insertion task, it finds the nearest gap and locks all thread that overlaps the shift distance. Once locked, the segments are shifted and inserted. For example, in Figure 6.8b, the insertion position is located in thd_1 while the closest gap is in thd_2. The meta locks both threads to carry out the insertion before releasing them. thd_3 and thd_4 can freely carry out their tasks during this process.

The meta thread also inserts new segments at the boundaries of partitions. Using the same example of thd_1 and thd_2, if the segment is at the last element of thd_1, it can also be the first element of thd_2. We allow thd_0 to decide the position that minimizes the shift distance.

Retraining SWmeta: The meta thread may trigger a retrain when carrying out the insertion task if (1) there are too few/many gaps, or (2) shifts generate too high of an error. SWmeta retraining requires locking all threads, as depicted in Figure 6.8c. However, we reduce the critical section by allowing thd_0 to conduct retraining based on a current snapshot of the data before blocking threads. Once the retraining is complete, the meta thread blocks all threads to establish a new SWmeta, before redistributing the segments.

6.6 Experimental Evaluation

We experimentally evaluate SWIX against current state-of-the-art indexes (serial and concurrent) across various parameters and workloads. We list the gain in performance from the evaluation below:

- For the general case under a standard IBWP (described in Section 6.6.1)
  - SWIX shows significant memory efficiency compared to the baselines. On average, it occupies 22.22% of the memory required for a B+Tree, 42.79% of that required for an IMTree, 12.06% of a PGM-index, and 25.91% of ALEX.
  - SWIX does not compromise in latency, showing a speedup of 1.30× over B+Tree, 1.20× over IMTree, 1.59× over PGM-index, 1.19× over ALEX, and a notable 136.55× over a queue.

- For the concurrent workload, Parallel SWIX demonstrates a memory saving by using 41% and 34% of the memory required by FINEdex and XIndex, while achieving up to a 1.90× and 3.45× higher throughput compared to these two baselines for update-heavy workloads.

6.6.1 Experimental Setup

Dataset & Setup: We implement SWIX and conduct all experiments in C++ using the `gcc` compiler on a 16-core machine with Intel E5-2690@2.60GHz and 64GB of memory running Ubuntu 14.04.

The dataset comprises eight datasets for the SOSD benchmark [108]. Specifically, four synthetic (*lognormal*, *normal*, *uniform dense* (*udense*), and *uniform sparse* (*usparse*)), and four real-world (*books*, *fb*, *wiki*, and *osm*). The datasets are designed to assess index performance under different data distributions, which is crucial for the performance of learned indexes. We summarized the results of all datasets in Figure 6.11. For the comparative analysis, we focus the discussion on two opposite datasets: *udense*, characterized by its linear distribution at both local and global scales, and *osm*, which exhibits a non-linear distribution at both scales.

However, the SOSD data only contain one-dimensional key values (which are shuffled to ensure random updates). To simulate the arrival and departure rates of streaming data, we generate timestamps to simulate both uniform and skewed update patterns. This mimics the YSB [42] and Stock [214] streaming benchmarks. Note that we opt not to include these two benchmarks as they distribute timestamps uniformly. Our workload is more exhaustive in evaluating the effect of data bursts on index performance.

Baseline: The baselines are presented in Sections 2.3.3 and 2.4.4. To recap, these baselines include an array-based queue, STX B+Tree [29], the dynamic PGM-index [92], ALEX [134], XIndex [179], and FINEdex [178]. We grid-searched the fanout for the B+tree and the error in PGM from 4 to 4096 to select the configuration that yields the best latency in each test. A secondary index is integrated with these indexes to support IBWP workloads. Specifically, we

included a timestamp-to-key mapping (in a sorted queue) to find the keys of expired timestamps. To ensure fair comparison based on index performance, we did not include the search times and memory sizes of the secondary indexes in our results.

Additionally, we include the STX B+Tree implementation of IMTree [162] and FLIRT (Chapter 5) as streaming indexes. The insertion tree has a size of $0.15n$ where n is the data size based on the observations from the authors [162]. The parallel version of IMTree is not included due to the unavailability of the code. Note that we compare FLIRT in the ordered workload.

Default Workload: The IBWP workload is simulated with a sliding time window. The amount of data in the window is determined by the window length W. It is worth noting that enqueues don't necessarily happen at every timestamp; some timestamps may receive multiple enqueues while some timestamps may receive none. Therefore, the data size n in the window fluctuates over time (with a positive correlation to W).

The default workload starts with an initial W of 10 million and performs 40 million iterations. The initial W has one data per timestamp to unify the data size and the time window such that $n = W = 10M$. Each interaction includes an update that moves the window by one step (in terms of time) and a search query. We use a range query with a scan range of 1000 keys as the standard search query.

For a uniform update workload (default case), we distribute the timestamps uniformly to ensure the average data size during updates equals W. For the skewed workload (varying arrival and departure rates), we generate timestamps from a normal distribution $N(\mu, \sigma^2)$. The arrival and departure rates are controlled by changing σ^2. For example, to simulate a data burst, we use smaller variances. Therefore, in this case, n depends on σ^2, while W is held constant.

Performance Metrics: The units used to measure memory usage is MB, latency is measured in μs, and throughput is shown with ops/s. Memory usage includes index and data size, while latency and throughput are reported for one iteration (includes time for search and update). We time the execution time using a *Read Time-Stamp Counter* (RDTSC) that has less than 5% overhead [102, 214].

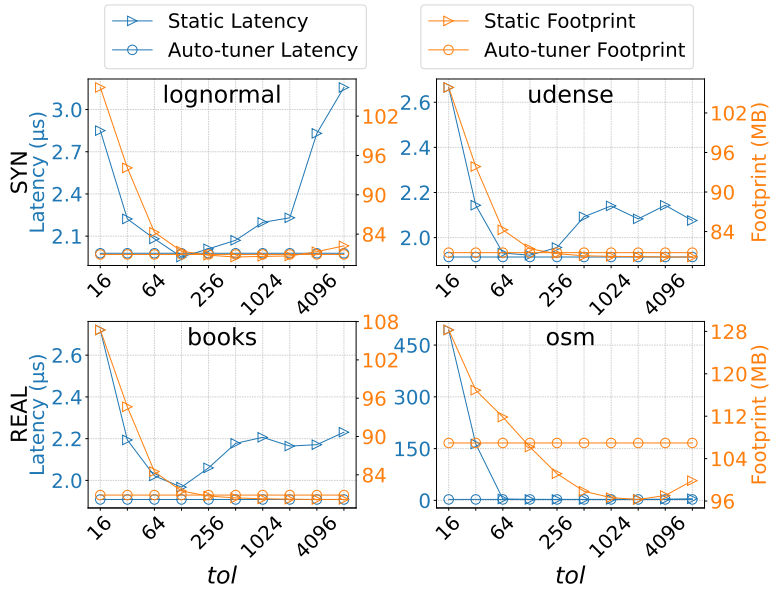


Figure 6.9: Comparison between auto-tuned and static *tol*.

6.6.2 Self Analysis

Evaluation of Auto-tuner: Before comparing SWIX with competitors, we first examine the performance of the auto-tuner. We executed SWIX with the auto-tuned and compared it against the case where we fixed the *tol* values (spanning from 16 to 4096). The auto-tuner tunes the index once every time 10% of the window progresses (every $0.1W$). We show the results in Figure 6.9, where the results for the synthetic datasets are shown in the top row, and the real datasets are shown below. Latency is represented with the blue lines (aligned with the left y-axis), while the total memory usages are displayed as orange lines (corresponding to the right y-axis). We use triangles to symbolize the static *tol* and circles for the auto-tuner.

We can observe a convex memory usage and latency shape with varying *tol*. This suggests an optimal point that minimizes memory and time for each data distribution, hence a reason to use the auto-tuner. The auto-tuner is shown to give near-optimal latency performance, which verifies the heuristic model's ability to estimate the time cost accurately. However, we may get a larger index size for datasets with high non-linearity. Specifically, the auto-tuner tends to trade space for time in *osm*. One improvement is to apply constrained optimization with a memory budge to limit the memory footprint for highly non-linear data. However, this comes at a cost of longer processing times. For now, we use our current auto-tuner, which is much

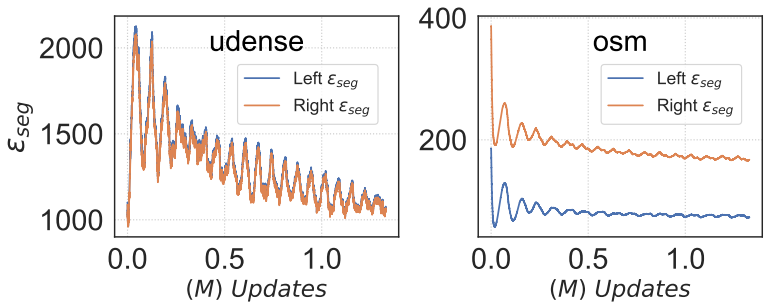


Figure 6.10: ϵ_{seg} evolution over updates.

faster and suitable for real-time requirements, and we can still achieve competitive memory savings against other indexes.

Analysis of Segment Error: Another detail before moving on to the comparative analysis is verifying if the segment error is reduced during distribution shifts. Figure 6.10 shows the average segment error with more updates. For both linear and non-linear datasets, the errors are shown to have a damped cosine wave pattern, which verifies the auto-tuner's ability to adapt to distribution shifts. Note that for the *udense* dataset, we get a relatively balanced left and right error. However, we can see that two errors differ in the face of *osm*. A larger right error suggests that segments generate more gaps towards the right end of the array, which causes more right shifts. Furthermore, the absolute error value reveals that the segments are generally smaller for non-linear datasets, which verifies the auto-tuner's tendency to trade space for time.

6.6.3 Summary of Comparative Analysis

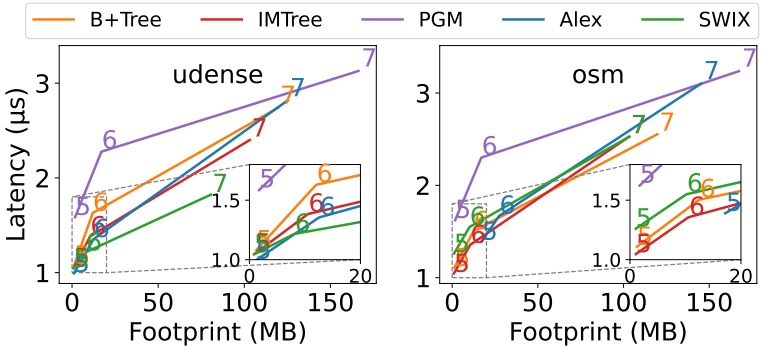
We summarize the comparative results in Figure 6.11. The most time-efficient indexes are highlighted in dark green, while the most memory-efficient indexes are highlighted in light green. For most datasets, we can observe SWIX to be both memory and time-efficient. Notably, SWIX is shown to be memory efficient at large datasets as the overhead at MAX usage can be less than the AVG memory consumption.

We now analyze the results in detail, starting with memory usage. Tree-based structures, represented by the B+Tree, tend to consume more memory due to their hierarchical structure. Using multiple trees, in the case of the IMTree, is shown to reduce memory consumption by

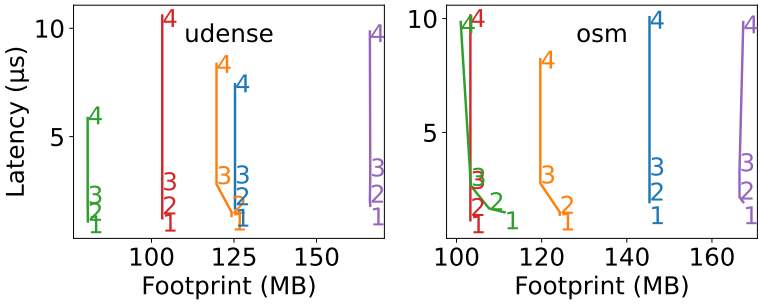
Index Type	Index	Metrics	SYN								REAL							
			lognormal		normal		udense		usparse		books		fb		osm		wiki	
			AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
No Index	Queue	*Overhead (%)*	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		Latency (µs)	2857.8	5664.1	2794.4	5537.4	2809.8	5568.9	2793.1	5534.1	2790.6	5532.4	2817.2	5583.9	2807.4	5562.4	2840.7	5630.4
Alogrithmic Index	B+Tree	*Overhead (%)*	72.2	130.7	67.1	130.5	66.9	130.5	72.3	130.7	72.2	130.7	66.9	130.5	67.1	130.5	66.9	130.5
		Latency (µs)	27.1	268.2	26.3	259.1	27.7	274.1	27.0	266.2	27.1	265.9	26.5	261.4	26.2	255.3	26.9	265.94
Streaming Index	IMTree	*Overhead (%)*	35.2	54.8	35.2	54.8	35.2	54.8	35.2	54.8	35.2	54.8	35.2	54.8	**35.2**	**54.8**	35.2	54.8
		Latency (µs)	25.3	246.9	24.9	242.5	24.7	240.1	24.6	239.9	24.6	240.8	**24.4**	**238.6**	**25.7**	252.7	24.5	238.0
Updatable Learned Index	PGM	*Overhead (%)*	124.9	143.8	124.8	143.8	124.4	142.0	124.5	142.0	125.0	144.6	125.4	144.0	125.3	143.4	125.0	144.2
		Latency (µs)	32.6	315.6	33.1	321.0	32.3	312.8	32.4	312.7	32.6	316.0	33.3	323.3	33.5	323.7	32.7	315.6
	ALEX	*Overhead (%)*	45.6	56.7	43.4	55.3	55.9	64.1	55.9	64.1	56.0	64.1	50.0	53.1	102.8	111.3	40.5	47.3
		Latency (µs)	19.4	188.8	**18.6**	**180.8**	28.0	282.0	20.1	195.7	24.7	245.1	32.2	317.3	31.4	310.2	24.7	247.1
Streaming Learned Index	SWIX	*Overhead (%)*	**7.0**	**6.3**	**7.2**	**6.5**	**6.8**	**6.1**	**7.0**	**6.2**	**6.8**	**6.1**	**32.4**	**48.9**	39.1	56.3	**14.3**	**19.0**
		Latency (µs)	**19.2**	**186.5**	19.6	190.9	**18.8**	**182.4**	**18.5**	**179.0**	**18.3**	**176.8**	25.0	245.0	26.0	**252.6**	**22.8**	**224.2**

Figure 6.11: Memory overhead $\frac{total\ size - data\ size}{data\ size}$ and latency $\frac{total\ time}{total\ updates}$ for each experiment setting excluding the ordered workload. For memory overhead, MAX is measured when the total memory consumption (*total size*) is at its largest and can be smaller than the AVG.

reducing the tree's height while having fuller trees. The search tree of the IMTree is shown to be 99% filled, while the insertion tree is 71% filled, outperforming a taller B+tree with a 63% occupancy. However, having too many sub-indexes has been shown to have an adverse effect on the PGM's log-merge tree design. Even if adding a learned model (in ALEX) has been shown to be a viable strategy to reduce memory usage from B+trees, the PGM-index still has the highest average memory usage out of all indexes. The main inefficiency of ALEX comes from the split mechanism. To reduce retraining cost, ALEX expands to be 60% full, meaning each node contains 40% gaps. Furthermore, ALEX only allows splitting by power of 2 to ensure the top-down approach does not generate error in the internal node. This leads to wasted space in the internal nodes, where multiple pointers can be made to the same child node. In contrast, SWIX maintains a single internal node (SWmeta) that only allows most 20% of gaps (reduced to 5% upon retraining). We retrain and split segments based on the data distribution, which avoids data fragmentation to ensure the minimum number of segments. Both SWIX and ALEX show that compaction efficiency from learned models depends on data distribution. Learned models cannot achieve much compaction for hard-to-learn distributions such as *osm*. However, in this case, we show that combining a two-level structure to learned models avoids creating an index larger than traditional indexes (as in the case of ALEX).



(a) Time (μs) and space (MB) trade-off with window sizes.



(b) Time (μs) and space (MB) trade-off with scan ranges.

Figure 6.12: Analysis of different window sizes and scan ranges.

In terms of latency, learned indexes are shown to be able to reduce the search space for distributions that are not difficult to learn. This makes them data-dependent compared to the B+tree and IMTree, which are distribution agnostic. Interestingly, the PGM's log-merge structure is shown to smooth out the distribution and achieve consistent performance across datasets. However, the magnitude of the latency makes PGM unappealing even against traditional indexes. ALEX meets the expectation of a learned index. Compared to SWIX, the disadvantages come from its increased height (as nodes are allowed to split downwards) and unbalanced tree structure. The main message here is that the two-layer structure of SWIX does not compromise the speed-up benefits from learned indexes while ensuring a small size.

6.6.4 Window Length

We now evaluate the performance of the indexes for different stream parameters starting with the window length W. Changing the window length simulates different data sizes. Figure 6.12a shows the results when varying W from $100K$ to $100M$. We labeled the window lengths

according to their exponent values $(10^5, 10^6, 10^7)$. The main highlight from the results is that SWIX shows the highest latency-to-size ratio, indicated by its proximity to the bottom left corner of the figures.

The advantages of SWIX become increasingly clear with larger W compared to other learned indexes. However, as mentioned in Section 6.6.3, the advantages are data-dependent. For *udense*, the average segment level exponential search length only increases by 1.36× as the window length increases while the number of segments grows by roughly 4.85×. This is marginal compared to the data size increasing 100×, which explains the performance gains. However, for *osm*, the auto-tuner tries to minimize the latency while non-linearity increases with larger window lengths. We are able to achieve an average exponential search length increase of 1.66× (about 22% increase compared to *udense*). However, this results in a 63× increase in segments for the same 100× data size increase. The retrain length also increases by 34×, which results in a higher total latency. We can observe that the performance (space and time) of SWIX is error-bounded rather than space-bounded, which allows it to scale exponentially well for easy-to-learn distributions.

For hard-to-learn datasets, a traditional index such as a B+tree or IMtree may be preferred for more consistent performance. PGM and ALEX are shown to both suffer at large W. ALEX is especially dependent on data distribution. When the window length is $100M$, it generates 657 model/internal nodes, compared to the 232 internal nodes of the B+tree. This results in a very deep and unbalanced tree. ALEX also unexpectedly performs poorly for *udense* compared to traditional indexes. Interestingly, the difference between ALEX and SWIX highlights the performance gains of our design since both structures have two levels for *udense*.

6.6.5 Scan Range

We now evaluate the performance for short and long-range queries by adjusting the scan range from 10^1 to 10^4. The results are shown in Figure 6.12b, and we use the exponential values to label the scan ranges. The vertical lines mean the memory footprint does not change with the scan range (B+tree differs in size due to grid searching the fanout for optimal latency). We use

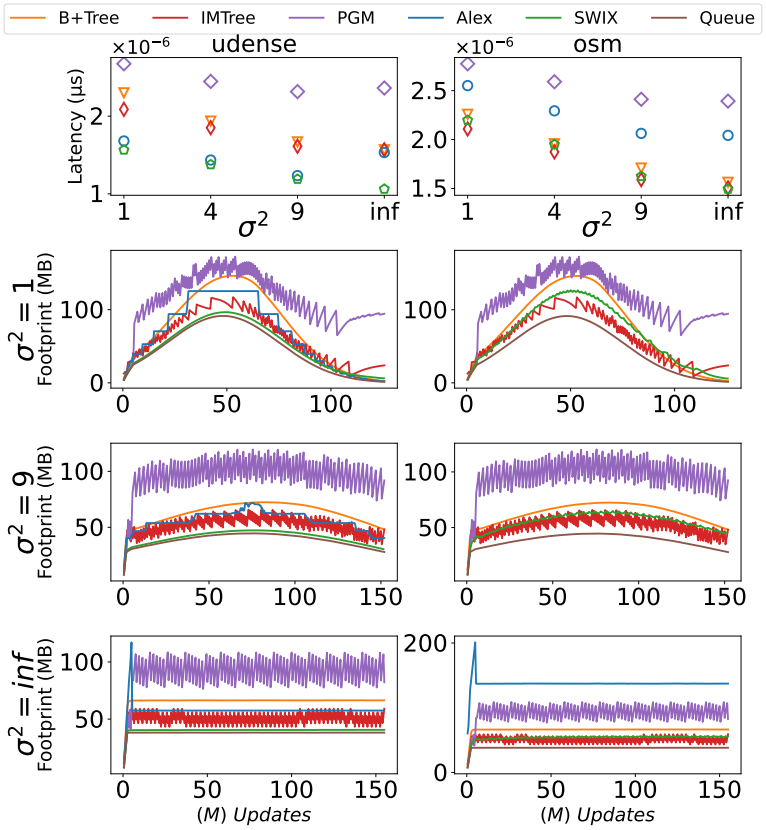


Figure 6.13: Latency (μs) and Index footprint (MB) for different degrees of σ^2 in timestamps.

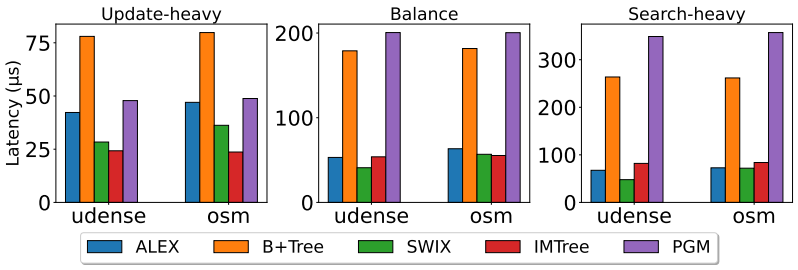
this experiment to highlight the limitation of the implicit deletion strategy, where the memory overhead increases as fewer keys are examined during scans. The drawback is more noticeable for hard-to-learn datasets due to having more segments. The results also reveal that ALEX and PGM's design goal is speed-focused rather than size-focused.

6.6.6 Skewness in Timestamp

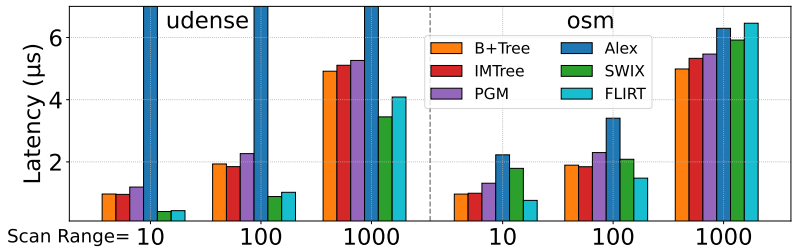
We evaluate the indexes' ability to handle the sudden increase in data volume common in streaming workloads. This is known as a data burst, where the insertion rate exceeds the deletion rate, resulting in a rapid increase in data size. Subsequently, the data size quickly decreases back to the pre-burst size with a higher deletion rate. This experiment can give us insight into the index's ability to scale with changing data size during runtime. We simulate various data burst rates with the timestamps generated in Section 6.6.1. The results are shown in Figure 6.13 where we vary σ^2 from 1 to inf. A smaller value of σ^2 indicates a data burst

while $\sigma^2 = \text{inf}$ represents a uniform time distribution (default case). The time performance is shown in the top two rows. We show the temporal progression over updates for $\sigma^2 = 1, 9, \text{inf}$ for memory usage. Note that ALEX is unable to complete the memory test for *osm* at $\sigma^2 = 4, 9$.

The highlight is the smooth memory profile of SWIX during shifts in data size. This provides some insight into why the MAX overhead is close to the AVG overhead seen in Figure 6.11. We attribute the smooth memory profile to the dynamic structure of SWarray, which allows the structure to adjust at run-time according to the data distribution. This differs from fixed structures of the baselines. For example, ALEX splits the structure as powers of 2, which tends to reserve more space, as shown by the step-wise change in memory usage and the spike in the $\sigma^2 = \text{inf}$ case. We would also like to mention that the saw-like spiky pattern for PGM and IMtree results from batching deletes. In both indexes, data expires during merges, which means the index is not compacted until the number of inserts reaches the merge criteria. This effect is exemplified by the $\sigma^2 = 1$ case, where the merge criteria are never reached at the end of the data burst.



(a) Varying read-write ratio.



(b) Ordered data.

Figure 6.14: Comparison of learned indexes on different workloads.

6.6.7 Read/Write Ratio

We now study the performance of indexes for two workloads (read-write ratio and ordered data) and start with the read-write ratio workload. Figure 6.14a presents the results for three read-write ratios: An update-heavy workload of an RW ratio of 1:10. a balanced case at 1:1 and a read-heavy workload at 10:1. We exclude the memory usage figure as it is identical to the previous analysis. We standardize the number of updates at $400M$ as an anchor. Therefore, the read-heavy workload will have $10\times$ more reads compared to the balanced workload. Similarly, the number of reads is a magnitude smaller for the update-heavy workload compared to the balanced workload. By comparing the three workloads, we can observe that search is the most costly operation. SWIX benefits from the learned model for search and dynamic SWarray for updates to achieve the lowest average latency. The IMtree is shown to be the most updatable index, as it does not have a model to retrain and maintain. The drawback is the reduction in search space from learned models.

6.6.8 Ordered Workload

Figure 6.14b presents the results for the ordered workload for different scan ranges, where the dataset is organized sequentially and updates are in the form of enqueues and dequeues. The memory overhead for *osm* are 62.2% for SWIX, 87.3% for ALEX, 119.3% for PGM, 100.4% for B+Tree, 46.2% for IMtree and 22.9% for FLIRT[§]. This is the same workload in Chapter 5; we omit the analysis of baseline indexes, which can be found in Section 5.5.

SWIX is shown to perform similarly to FLIRT but suffers from *osm*. The auto-tuner is designed to optimize performance for random insertions across the entire index, which means it cannot capture the distribution change occurring in the last segment very efficiently. Furthermore, the implicit deletion strategy becomes less effective, as not all searches happen at the start of the index, which allows expired data to accumulate.

[§]FLIRT's overhead is calculated over a 1D dataset as FLIRT does not store the key and timestamp separately.

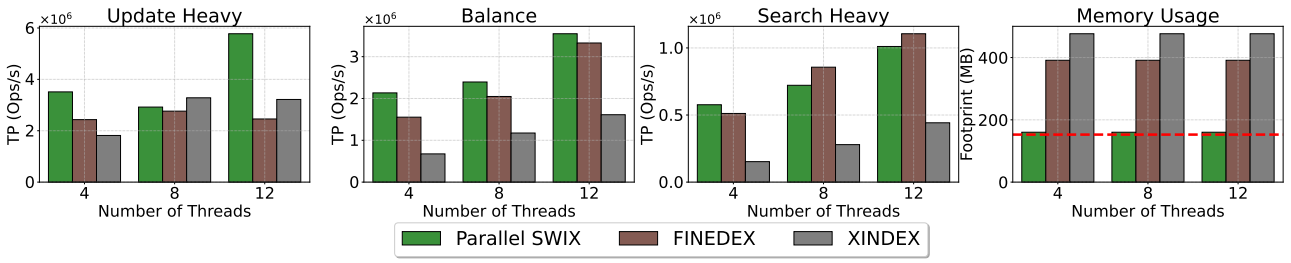


Figure 6.15: Comparison of parallel learned indexes with random insertion. The first three figures (from the left) indicate the throughput for different read-write ratios. The fourth figure shows the memory usage.

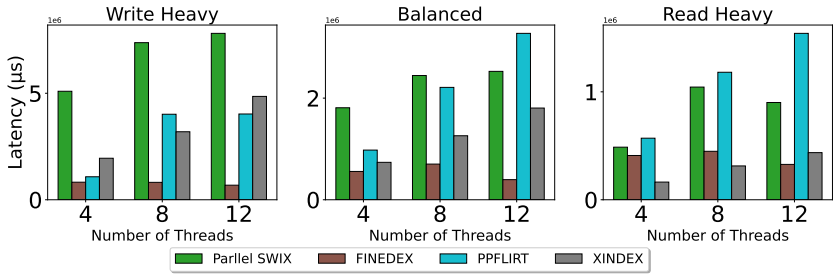
On the other hand, FLIRT ensures the error in each segment, allowing it to achieve superior search performance even for *osm*. However, this also means that FLIRT generates smaller segments, increasing the scan cost from pointer traversals. Combining both memory and speed, FLIRT is shown to be the best design for an ordered workload since it uses a two-level design and does not have any gaps or buffers.

6.6.9 Parallel Learned Indexes

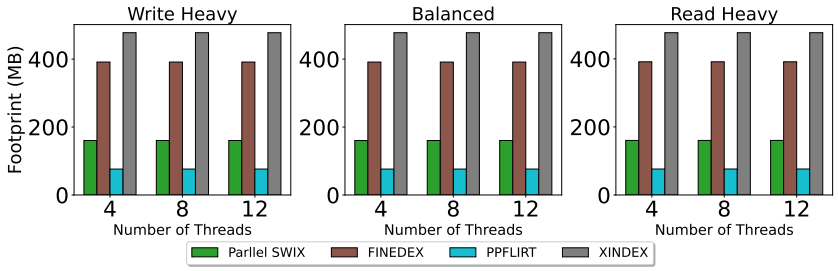
Random Insertion: We evaluate the performance of parallel indexes in Figure 6.15, where we show the scalability (in terms of throughput) as bar charts for different read-write ratios in the first three figures. The memory usage is shown in the last figure, which is the same for all three read-write ratios. For context, the data size is shown as the red dashed line.

Both XIndex and FINEdex employ a concurrent framework allowing threads to access a global index. Parallel SWIX adopts a parallel structure where each thread manages a subset of data. This approach reduces the load in each process and shows a better search performance compared to concurrent indexes for a low thread count. The ideal case is when the threads can work independently to fully utilize the potential of the parallel design, as shown in the update-heavy workload. However, the efficiency decreases when the workload accesses multiple partitions and requires each thread to carry out the same work (e.g., range queries). In this case, a concurrent design scales better, as shown by FINEdex out-scaling Parallel SWIX.

In terms of design differences, although all indexes have "two" layers, FINEdex incorporates a



(a) Throughput.



(b) Memory Usage.

Figure 6.16: Comparison of parallel learned indexes with ordered insertion.

B-tree to store the prediction models at the root level while using a modified B-tree to store data at the node level. The modified B-tree can be viewed as data level buffers that keep the leaf level sorted to improve search performance, as seen in the search-heavy workload. XIndex's root layer includes an RMI with a pointer to the leaf level. Within the leaf level, there is an array of linear models and a buffer for insertions. Both SWIX and XIndex use a buffer that increases the search time compared to FINEdex, especially for longer scan ranges. However, the buffer is shown to be much more efficient for updates.

Regarding memory usage, Parallel SWIX is significantly more memory-efficient than the two baselines. Although FINEdex's data level B-trees have an average of 82% occupancy, the entire structure comprises several B-trees, adding to the memory overhead. One reason for XIndex's large size is that it generates temporary buffers during retraining.

Ordered Insertion: In our final comparison, we evaluate the ability of the indexes to handle an ordered workload. The results are shown in Figure 6.16 including PPFlirt[¶]. FINEdex suffers heavily from this workload because all threads wait in line to modify the first and last segments during updates. Under the same design, XIndex's buffer greatly helps with update performance. PPFlirt is shown to perform well for search-heavy workloads since the data is

[¶]We include PPFlirt because it outperforms PSFlirt in a uniform query workload (Details in Section 5.5.9).

kept sorted and does not have gaps. However, PPFlirt suffers from an update-heavy workload because the update threads need to move between threads when the enqueue partition is full. Parallel SWIX, on the other hand, does not require any thread movement.

To summarize, Parallel SWIX stands out for its minimal memory overhead and competitive search performance against FINEdex and XIndex under random insertions. Its parallel design is most effective when tasks are independent between threads, which allows it to excel at update-heavy workloads regardless of whether the insertions are random or ordered.

Further analysis of Synchronization Overhead: Lastly, we want to discuss additional analysis that can profile the synchronization overheads in multi-threaded processing. The main source of synchronization overhead for Parallel SWIX comes from the meta thread locking the root level during updates. This can occur when insertions are at the boundaries of two partitions, when a partition is full, or when the SWmeta is retrained. To measure the synchronization overheads in SWmeta, we can use lock wait and thread wait times for each thread (except the meta thread) to measure the impact of synchronization on the throughput of each thread. For the meta thread, we can measure the CPU usage time to review its utilization. Furthermore, we can use the meta queue wait time to profile the distribution of meta updates and look for hot spots. The experiments should be conducted under our uniform and skewed workload (assess the impact of distribution skews) with various read-write ratios (assess the impact of updates) for a holistic view of Parallel SWIX's performance.

6.7 Summary

In this chapter, we showcase SWIX, a lightweight learned index with analytical cost bounds designed for index-based window processing. SWIX utilizes a dynamic, lightweight, and parallelizable design based on a two-layer flat structure. It uses an auto-tuner to dynamically adjust its size and search performance based on shifting distributions at runtime. When the data distribution is unknown, we use a second derivative-based segmentation algorithm that ensures prediction accuracy while minimizing the number of segments. Lastly, we propose Parallel

SWIX, a parallel learned index that utilizes a partition-based parallel design to increase search and update performance with minimal synchronization.

6.8 Limitations of FLIRT and SWIX

Before concluding this thesis, we will discuss some limitations of the two updatable learned indexes: FLIRT (Chapter 5) and SWIX. We first provide an analysis of the limitations in terms of architecture (flat structure) before moving on to specific limitations and improvements for stream processing.

6.8.1 Limitations of Flat Index Structures

As discussed in the previous two chapters, a flat index structure + a learned model can improve memory efficiency while ensuring low latency compared to hierarchical structures. However, using this type of structure also has limitations.

The main limitation of the general flat index structure is its ability to scale with a high volume of data, especially when the data size grows during runtime. Since all records are at the same level, the index tends to grow horizontally, making the root node very large and increasing the response time during querying. This also becomes a problem when you try to parallelize the flat structure. Specifically, if the index grows horizontally, we must constantly repartition the index to ensure load balance, which generates overhead.

We considered these limitations when designing FLIRT and SWIX by incorporating learned models and a dynamically resizable structure. However, this introduces a new challenge, such as the reliance on the performance of the learned model to capture the data distribution. This is a common problem with instance-optimized design; since we expose our parameter space to allow the index to adapt to different workloads and data distributions, the algorithm finding the "optimal" parameter setting becomes crucial. In terms of a two-level structure, the performance of the learned model in the root model becomes very important. Although we

considered this problem while designing FLIRT and SWIX by using a bottom-up approach to adjust the structure in runtime, it may become a limitation with larger and more complex (in terms of data distribution) datasets. Specifically, we used linear models at the root level to balance retraining cost and accuracy; however, when the distribution becomes hard to learn (as in the case of *osm* in Table 6.11, we need to generate a lot of segments to ensure performance in each segment. This results in the root level becoming large, which may require multiple models to ensure accuracy[‖].

6.8.2 Limitations in Stream Processing

Although FLIRT and SWIX are designed to handle various streaming workloads, they are both designed as in-memory indexes. This is sufficient for the temporal filtering in range queries, joins, and aggregation. However, we run into some limitations for much more complex queries that require datasets that cannot fit into memory.

For the general case requiring larger-than-memory indexes, both FLIRT and SWIX can use chain indexing to maintain multiple sliding windows. The chaining can be done with respect to time spans or key ranges depending on the specific application. Here, we illustrate how chain indexing can be applied to two real-world case studies:

- *Historical + Recent Data Queries*: It is common for analytics tasks in many applications to require querying both historical and real-time data. For example, for performance evaluation in aviation and ride-sharing, average speeds (e.g., over the past 7 days) of a particular route/road are determined and compared with real-time data to evaluate the performance of the aircraft/vehicle. In this case, the chaining would be done in the time dimension, where we can keep building an index for each time window (e.g., 1 day period). The top-level chain is a time-ordered array of indexes (similar to FLIRT's top-level queue, without deletion). We can also apply an optimization since we know the historical data

[‖]As discussed in Section 6.6, for hard-to-learn distribution, we still would not recommend a hierarchical learned index (see performance of ALEX), instead a flat index with more models in the root level or traditional hierarchical structure (IMTree) is preferred.

is static by compressing the gaps in SWIX once the most recent time window is no longer available. The most recent data is always kept in memory during runtime, while historical data can be fetched for secondary storage (hard drive, distributed store).

- *Very Large Window Sizes*: Another case would be applications with very large sliding windows (e.g., millions of records over days or weeks) or very dense (e.g., very frequent updates in a short period). An example would be financial transaction streaming for fraud detection, which may require analyzing millions of transactions per hour. In this case, the top-level chain index can be built over time ranges (similar to the example above) or key ranges depending on the application. For example, key ranges may be more applicable to applications where queries are based on the primary key rather than timestamp filtering.

Note that the specific implementation depends on the specific query (e.g., how much historical data is required) and the system configuration (e.g., shared memory or distributed memory architectures). However, the data cannot fit in a single machine's memory in both applications. Therefore, one of the biggest challenges is to reduce I/O cost (biggest source of latency) when fetching data from other storage mediums. Furthermore, to support distributed computing, we must also consider distributing the chain index across machines or introducing fine-grained locks in each index in the chain. For example, for a simple case where the data cannot fit in memory, a potential solution would be to build a prefetcher to obtain blocks of data at once and to partition the chains such that each partition contains a range of continuous days.

Chapter 7

Conclusion

7.1 Summary of Thesis Achievements

In this thesis, we proved our hypothesis and showed that a multi-layer, modular approach to designing index structures using both traditional and learned indexing achieves better scalability and flexibility compared to existing approaches. Specifically, our insight into the pattern of trajectory updates reveals that the spatial updates generally remain within predetermined regions while temporal updates continuously grow, adding and removing data points over time. Therefore, our proposed modular approach combines the robustness of traditional index techniques (for managing spatial distribution changes at a global level across multiple machines) and the speed up from learned indexes (for filtering temporal data whose data distribution is more predictable in a local setting). Formally, we answer two questions from different perspectives to provide a holistic solution. The two questions are: (1) what traditional indexing techniques can improve spatial updates in trajectory indexes and preserve continuity across multiple machines? (2) How can learned indexes efficiently adapt to the update heavy workload of temporal updates in each machine?

In Chapter 3, we answer the first question regarding trajectory distance joins. We propose a distributed approach to answer distance joins efficiently while minimizing the impact of updates on data partitioning. The challenge was to account for incoming data and changes in join

predicates (e.g., spatial and temporal range). We tackle this challenge by introducing a uniform grid partitioning strategy and in-partition grid that correlates its size with the join predicates. Additionally, we allow trajectories to preserve continuity by storing them as a sequence of line segments that support interpolating information between points. Our approach has been shown to improve efficiency and scalability through extensive experiments by achieving a 43× increase in processing time compared to the state-of-the-art.

In Chapter 4, we tackle the first question for a privacy-preserving range query in a distributed network. We propose an in-network framework that allows queries to be answered in the sensor network. In-network systems reduce the overall communication cost from continuous updates and ensure privacy through data locality. However, they are not designed for query processing and suffer from two distinct problems (double counting and dead-space problems). To achieve efficient query processing, we propose to select a subset of sensors based on both sensor and query distribution. We capture the sensor distribution through a planar graph representation of the sensor network. We then developed sensor selection techniques for both known and unknown query distributions. A constant-sized machine learning regression model compacts the counts in the temporal dimension and allows queries with arbitrary time intervals, leading to the second question in the next chapter. The framework was evaluated on real-world mobility data and yields a relative error of at most 13.8% with 25.6% of sensors while achieving a speedup of 3.5×, 69.81% reduction in sensors accessed and a storage reduction of 99.96% compared to using the entire sensor network.

In Chapter 5, we move on to the local level and focus our attention on updates in the temporal dimension (second question). Our in-network framework revealed that although a constant-sized regression model can use a buffer and retrain periodically to accommodate updates, this strategy is inefficient for this streaming-like workload. Therefore, we treat the updates in the temporal dimension as a streaming workload. The challenge was ensuring stream indexes' updated efficiency while having similar search performances to search-optimized structures. We propose FLIRT, a queue-based learned index that accelerates search through piecewise linear models (known as linearly predictable segments). It is specifically designed for the workload of a queue where keys are time-ordered. We designed FLIRT to automatically optimize its

search performance to accommodate data distribution shifts from updates. Our comparison with existing indexes shows that FLIRT can achieve up to 13× speedup over B+Trees, up to 7× speedup over queues, and up to 109× speedup over updatable learned indexes. To further improve search performance, two multi-threaded variants of FLIRT were proposed for uniform and skewed query workloads, which can boost FLIRT's throughput up to 32×.

Finally, we extend the problem to a general streaming query workload where keys can be inserted randomly in Chapter 6, which allows querying by trajectory IDs while updating based on timestamps. This is known as an index-based window processing (IBWP) workload. Our objective is to create a memory-efficient index without comprising search and update performance. We propose SWIX, a lightweight index based on an update-optimized two-level structure combined with piecewise linear models to improve search. The two-level design reduces duplicates in the internal structure to minimize memory overhead. To accommodate for the height decrease, we designed a node structure that dynamically adjusts to the data distribution during updates. The result is a lightweight, dynamic, and parallelizable index that is experimentally shown to only occupy 22% to 42% of the size compared to state-of-the-art serial approaches while achieving up 1.2× - 1.6× average speedup over them. In terms of parallel performance, Parallel SWIX is able to increase the throughput by 3.45× with only 34% of memory consumption.

7.2 Future Work

Our work demonstrates the opportunities and promises of bridging ML-enhanced techniques and trajectory indexes; there are many optimizations that can improve trajectory and learned indexes. We highlight some directions for future research:

Incorporating Local Optimizations to the Global Level: In Chapter 5 and 6, we proposed FLIRT and SWIX as local optimizations within a single machine. Extending the two optimizations into a global setting across multiple machines is a direction for future work. Specifically, this may be useful for applications that require high scalability and throughout or geographically distributed systems (e.g., in-network).

Two directions exist for adapting the two learned indexes into a distributed setting. The first method is to adapt the index structure to account for partitioning, coordination/concurrency control, loading balancing, and cross-node communication complexities. For example, to adapt SWIX to a distributed setting, we would instead build a SWIX in each partition (similar to PPFlirt) instead of using a shared root node to avoid the bottleneck at the root level during querying. We can use a coordination service (e.g., ZooKeeper [101]) to route the data to the partition it belongs to. When the partitions become unbalanced, which requires splitting or merging of partitions, we need to consider the load balance in terms of the model error rather than the number of keys. This is the main challenge, as we need to consider the distribution of data and updates across partitions.

The second direction is to integrate the indexes into existing distributed streaming systems such as Apache Flink, Kafka, Amazon Kinesis, and Google Cloud Dataflow. Compared to the previous direction, this method is more flexible, with a trade-off of being less optimized for specific workloads and system configurations. For example, to integrate SWIX and FLIRT into Flink, we can implement them as map transformations. However, we must also consider the overhead from the state management data partitioning and the trade-off between scalability and low-latency performance.

Learning the Spatiotemporal Dimension: Although attempts have been made to learn the spatial dimension mainly through dimension reduction or projection: the common approach is to use a Z-ordered curve to transform the spatial problem into a one-dimensional problem and apply a learned index to the one-dimensional space. However, this approach is not ideal for combining the temporal dimensions as the spatial and temporal dimensions are not correlated (as we have seen). Furthermore, this tends to make the index very sparse. We suggest an alternative approach to learning the data ordering based on the data distribution, which minimizes the dead space from space-filling curves. Such a function can "learn" the distribution in the original space rather than in a projected space.

Updatable Trajectory Learned Index for Streams: A natural extension of Chapter 5 & 6 is to index trajectories explicitly. The challenge here would be to preserve the continuity of

trajectories during updates. For example, existing work tries to store points that belong to the same trajectory but are not spatially close. Instead, we can store by spatial closeness and use a prediction model to locate the points from the same trajectory. The challenge here would be that n trajectories may result in n models. Therefore, we need to devise a spacing scheme for points in the same trajectory that reduces the number of models and minimizes the index size.

Semantic Trajectory Textual Queries: Combining location and textural data has been widely studied in spatial search queries. In terms of trajectories, an example kNN query would find the k closest trajectories that contain a set of keywords given a query location. This query has three aspects: spatial, temporal, and textual. Semantic meanings have been added to trajectories' spatial and temporal components that implies the activity being carried out [204]. For example, points may be denoted "park" if the user walks into Hyde Park. Semantic meanings have recently been added to the textual data to improve the accuracy over exact keyword matching [187] with word embedding and high dimensional vectors. It may be interesting to combine the two into the semantic trajectory and keyword queries, which allows queries to explore the context and meaning of trajectories and keywords (in addition to spatiotemporal relationships). For example, find runners who like to jog in Hyde Park and their favorite closest coffee shops. However, the challenge is efficiently indexing high-dimensional (text) and low-dimensional (spatiotemporal) vectors.

Learned Indexes for Wireless Sensing Networks: Learned indexes have been extended to new hardware, such as persistent memory [215], non-uniform memory access (NUMA) [45], and GPUs [123]. Due to the inherent design goal of learned indexes to be lightweight and fast, one interesting area to explore (in terms of novel hardware) is indexing in wireless sensing networks (WSNs). We can combine learned indexing techniques with WSNs in two areas: (1) in-network indexing and (2) indexing within sensors.

The first problem deals with in-network indexing, where the index guides the path to the sensor nodes holding the data (local to each sensor or grouped at a "home" node) [84] as well as organizing data in the network to achieve minimum energy and communication cost. We can introduce learned-based techniques to model the data distributions and predict the

access patterns based on historical datasets. This can reduce the number of access compared to static access patterns. We can further improve the index by learning multi-modal data access patterns. For example, a query requiring temperature, humidity, and luminance may need to access different sensors or motes.

The second problem deals with reducing storage and computation costs in each sensor, which is crucial to battery-operated energy-constrained sensors. We have shown the case for improving `count` aggregate queries in Chapter 4. There are other aggregate queries, such as `max` and `min`, where we can replace a model to replace exact values. We can combine this with in-network indexing to reduce communication costs further. For example, we can store models that describe the attributes of lower-level nodes at higher levels to reduce accessing the full spanning tree (for a TAG system [126]). Given a communication constraint, we can decide to find the exact results or provide an approximate result, where the models also add a level of privacy. Additionally, providing differential privacy bounds for learned indexes would also be an interesting topic.

Acronyms

kNN k-Nearest Neighbor. xv, xxii, xxiii, 38, 40, 74, 81, 87, 93, 161

AI Artificial Intelligence. 28, 29

AI4DB Artificial Intelligence for Databases. 28–30

AIS Automatic Identification System. 55, 56

CAS Compare-and-Swap. 19

CDF Cumulative Distribution Function. xxi, 30–33, 86

CPU Central processing unit. 9, 19, 55, 153

DB4AI Database Techniques for Artificial Intelligence. 28

DBMS Database Management System. 8, 9

EDBT The International Conference on Extending Database Technology. 7

FIFO First In, First Out. 20

GDPR The General Data Protection Regulation. 2

GPU Graphics Processing Unit. 16, 161

HDFS Hadoop Distributed File System. 41

I/O Input/output. 9, 21, 41, 156

IBWP Index-based Window Processing. xxiv, 122, 124, 125, 140–142, 159

LP Linearly Predictable. 99–101, 105

LSM Log-Structure Merge. 20, 34, 36

MBB Minimum Bounding Box. 23, 24, 47, 49, 53, 60

ML Machine Learning. i, 19, 159

NASA The National Aeronautics and Space Administration. 1

NUMA non-uniform memory access. 161

RDD Resilient Distributed Dataset. 41

RDMA Remote Direct Memory Access. 16

RDTSC Read Time-Stamp Counter. 142

RMI Recursive Model Index. 33, 36, 152

RUSI Royal United Services Institute. 7

SIGMOD The International Conference on Management of Data. 7

SOSD Search on Sorted Data Benchmark. 111, 141

SQL Structured Query Language. 9, 41

STX Some Template eXtensions. 111, 141

WSN wireless sensing networks. 161

Bibliography

[1] OpenStreetMap on AWS. `https://aws.amazon.com/public-datasets/osm`. Accessed Apr. 30, 2020. (pg 88)

[2] Stx b+ tree revisiting binary search. https://panthema.net/2013/0504-STX-B+Tree-Binary-vs-Linear-Search/, 2013. (pg 101)

[3] Abel, D. J., and Mark, D. M. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems 4*, 1 (1990), 21–31. (pg 27)

[4] Abraham, I., Dolev, D., and Malkhi, D. Lls: a locality aware location service for mobile ad hoc networks. In *Proceedings of the 2004 joint workshop on Foundations of mobile computing* (2004), pp. 75–84. (pg 66)

[5] Agarwal, P. K., Gao, J., and Guibas, L. J. Kinetic medians and kd-trees. In *European Symposium on Algorithms* (2002), Springer, pp. 5–17. (pg 67)

[6] Aguilera, M. K., Golab, W., and Shah, M. A. A practical scalable distributed b-tree. *Proceedings of the VLDB Endowment 1*, 1 (2008), 598–609. (pg 19)

[7] Aji, A., Vo, H., and Fusheng, W. Effective spatial data partitioning for scalable query processing, arxiv e-print, 2015. (pg 21)

[8] Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., and Saltz, J. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. *International Conference on Very Large Data Bases* (2013). (pp 2, 41)

[9] Akaike, H. A new look at the statistical model identification. *IEEE transactions on automatic control 19*, 6 (1974), 716–723. (pg 136)

[10] Alarabi, L. *ST-Hadoop: A MapReduce Framework for Big Spatio-temporal Data Management*. PhD thesis, University of Minnesota, 2019. (pp 12, 16)

[11] Alarabi, L. Summit: a scalable system for massive trajectory data management. *SIGSPATIAL Special 10* (2019). (pp 1, 39, 42, & 46)

[12] Alarabi, L., Mokbel, M., and Musleh, M. ST-Hadoop: a MapReduce framework for spatio-temporal data. *GeoInformatica 22*, 4 (2018). (pp 42, 46)

[13] Alarabi, L., Mokbel, M. F., and Musleh, M. St-hadoop: A mapreduce framework for spatio-temporal data. *GeoInformatica 22* (2018), 785–813. (pp 17, 28)

[14] Allen, J. F. Maintaining knowledge about temporal intervals. *Communications of the ACM 26*, 11 (1983), 832–843. (pg 11)

[15] Alves Peixoto, D. A distributed in-memory database system for large-scale spatial-temporal trajectory data. (pp 3, 9)

[16] Amanatides, J., and Woo, A. A fast voxel traversal algorithm for ray tracing. In *EG 1987-Technical Papers* (1987), Eurographics Association. (pg 53)

[17] Apache Software Foundation. Apache hadoop, 2024. (pp 25, 39)

[18] Arulraj, J., Levandoski, J., Minhas, U. F., and Larson, P.-A. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment 11*, 5 (2018), 553–565. (pg 19)

[19] Athanassoulis, M., and Idreos, S. Design tradeoffs of data access methods. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 2195–2200. (pg 20)

[20] Authority, D. M. https://www.dma.dk, 2020. (pp 1, 55)

[21] Bachfischer, M., Borovica-Gajic, R., and Rubinstein, B. I. Testing the robustness of learned index structures. *arXiv preprint arXiv:2207.11575* (2022). (pg 31)

[22] Bakli, M., Sakr, M., and Soliman, T. H. A. Hadooptrajectory: a hadoop spatiotemporal data processing extension. *Journal of Geographical Systems 21*, 2 (2019). (pp 39, 42)

[23] Bao, J., Li, R., Yi, X., and Zheng, Y. Managing massive trajectories on the cloud. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2016), pp. 1–10. (pg 17)

[24] Bao, X., Li, H., Qin, L., Xu, D., Ran, B., and Rong, J. Sensor location problem optimization for traffic network with different spatial distributions of traffic information. *Sensors 16*, 11 (2016). (pg 67)

[25] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 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* (1990), pp. 322–331. (pg 24)

[26] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517. (pp 16, 22)

[27] Bentley, J. L., and Friedman, J. H. Data structures for range searching. *ACM Computing Surveys (CSUR) 11*, 4 (1979), 397–409. (pg 22)

[28] Bertino, E., and Sandhu, R. Database security-concepts, approaches, and challenges. *IEEE Transactions on Dependable and secure computing 2*, 1 (2005), 2–19. (pg 29)

[29] bingmann. Stx-btree, 2019. Last Accessed: 2023-12-28. (pp 111, 141)

[30] Binna, R., Zangerle, E., Pichl, M., Specht, G., and Leis, V. HOT: a height optimized trie index for main-memory database systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2018), ACM, pp. 521–534. (pg 20)

[31] Boeing, G. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems 65* (2017), 126–139. (pg 88)

[32] Botea, V., Mallett, D., Nascimento, M. A., and Sander, J. Pist: An efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica 12* (2008), 143–168. (pg 17)

[33] Braz, F., Orlando, S., Orsini, R., Raffaeta, A., Roncato, A., and Silvestri, C. Approximate aggregations in trajectory data warehouses. In *2007 IEEE 23rd international conference on data engineering workshop* (2007), IEEE, pp. 536–545. (pp 63, 66)

[34] Cai, M., and Revesz, P. Parametric r-tree: An index structure for moving objects. In *In Proc. 10th COMAD International Conference on Management of Data* (2000), Tata McGraw-Hill, pp. 57–64. (pg 67)

[35] Cai, Y., Xu, W., and Zhang, F. ikd-tree: An incremental k-d tree for robotic applications, 2021. (pp 66, 67)

[36] Cao, X., Chen, L., Cong, G., Jensen, C. S., Qu, Q., Skovsgaard, A., Wu, D., and Yiu, M. L. Spatial keyword querying. In *Conceptual Modeling: 31st International Conference ER 2012, Florence, Italy, October 15-18, 2012. Proceedings 31* (2012), Springer, pp. 16–29. (pg 10)

[37] Chakka, V. P., Everspaugh, A., Patel, J. M., et al. Indexing large trajectory data sets with seti. In *CIDR* (2003), vol. 75, Citeseer, p. 76. (pp 17, 26, & 27)

[38] Chang, C.-Y., Kuo, Y.-W., Xu, P., and Chen, H. Monitoring quality guaranteed barrier coverage mechanism for traffic counting in wireless sensor networks. *IEEE Access 6* (2018), 30778–30792. (pg 67)

[39] Chen, T.-S., Chang, Y.-S., Tsai, H.-W., and Chu, C.-P. Data aggregation for range query in wireless sensor networks. In *2007 IEEE Wireless Communications and Networking Conference* (2007), pp. 4127–4132. (pg 83)

[40] Chen, Y.-Y., Suel, T., and Markowetz, A. Efficient query processing in geographic web search engines. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006), pp. 277–288. (pg 10)

[41] Cheng, R., Kalashnikov, D. V., and Prabhakar, S. Querying imprecise data in moving object environments. *IEEE Trans. on Knowl. and Data Eng. 16*, 9 (sep 2004), 1112–1127. (pg 63)

[42] Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B. J., et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)* (2016), IEEE, pp. 1789–1792. (pp 111, 141)

[43] Cooper, J. New developments in the use and manipulation of maritime trajectory data, Dec 2021. (pg 7)

[44] Cudre-Mauroux, P., Wu, E., and Madden, S. Trajstore: An adaptive storage system for very large trajectory data sets. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (2010), IEEE, pp. 109–120. (pg 17)

[45] Cui, L., Yang, K., Li, Y., Wang, G., and Liu, X. Difflex: A high-performance, memory-efficient and numa-aware learned index using differentiated management. In *Proceedings of the 52nd International Conference on Parallel Processing* (2023), pp. 62–71. (pg 161)

[46] Dai, Y., Xu, Y., Ganesan, A., Alagappan, R., Kroth, B., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. From wisckey to bourbon: A learned index for log-structured merge trees. In *OSDI* (2020), pp. 155–171. (pg 111)

[47] Dan, T., C.Luo, Y.Li, B.Zheng, and Gu.Li. Spatial temporal trajectory similarity join. In *Web and Big Data* (2019), J.Shao, M.Yiu, M.Toyoda, D.Zhang, W.Wang, and B.Cui, Eds. (pg 40)

[48] Data., T. T. R. https://www1.nyc.gov/tlc, 2020. (pg 1)

[49] Davitkova, A., Milchevski, E., and Michel, S. The ml-index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020* (2020), A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, Eds., OpenProceedings.org, pp. 407–410. (pg 3)

[50] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113. (pp 25, 40)

[51] Deng, K., Xie, K., Zheng, K., and Zhou, X. Trajectory indexing and retrieval. *Computing with spatial trajectories* (2011), 35–60. (pp 16, 23, 25, & 26)

[52] Dhanabal, S., and Chandramathi, S. A review of various k-nearest neighbor query processing techniques. *International Journal of Computer Applications 31*, 7 (2011), 14–22. (pg 10)

[53] Ding, H., Trajcevski, G., and Scheuermann, P. Efficient Similarity Join of Large Sets of Moving Object Trajectories. In *International Symposium on Temporal Representation and Reasoning* (2008). (pg 40)

[54] Ding, J. *Instance-Optimized Database Indexes and Storage Layouts.* PhD thesis, Massachusetts Institute of Technology, 2022. (pg 3)

[55] Ding, J., Marcus, R., Kipf, A., Nathan, V., Nrusimha, A., Vaidya, K., van Renen, A., and Kraska, T. Sagedb: An instance-optimized data analytics system. *Proceedings of the VLDB Endowment 15*, 13 (2022). (pp 3, 29)

[56] Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D., and Kraska, T. ALEX: An updatable adaptive learned index. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2020), p. 969–984. (pp 34, 35, 36, 97, 99, & 125)

[57] Ding, J., Nathan, V., Alizadeh, M., and Kraska, T. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment (PVLDB) 14*, 2 (Oct. 2020), 74–86. (pg 3)

[58] Ding, X., L.Chen, Y.Gao, C.Jensen, and H.Bao. Ultraman: a unified platform for big trajectory data management and analytics. *VLDB 11* (March 2018). (pg 1)

[59] Ding, Y., Li, Y., Zhou, X., Huang, Z., You, S., and Luo, J. Sampling big trajectory data for traversal trajectory aggregate query. *IEEE Transactions on Big Data 5*, 4 (December 2019), 550–563. (pg 88)

[60] Dittrich, J., Nix, J., and Schön, C. The next 50 years in database indexing or: the case for automatically generated index structures. *Proceedings of the VLDB Endowment 15*, 3 (2021), 527–540. (pg 29)

[61] Dittrich, J., and Seeger, B. Data redundancy and duplicate detection in spatial join processing. In *International Conference on Data Engineering* (2000). (pg 52)

[62] Dziedzic, A., Wang, J., Das, S., Ding, B., Narasayya, V. R., and Syamala, M. Columnstore and b+ tree-are hybrid physical designs important? In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2018), pp. 177–190. (pg 18)

[63] Egenhofer, M. J., and Mark, D. M. Modelling conceptual neighbourhoods of topological line-region relations. *International journal of geographical information systems 9*, 5 (1995), 555–565. (pg 12)

[64] Eldawy, A., and Mokbel, M. Spatialhadoop: A mapreduce framework for spatial data. In *International Conference on Data Engineering* (2015). (pp 39, 41)

[65] Eldawy, A., Mokbel, M. F., Alharthi, S., Alzaidy, A., Tarek, K., and Ghani, S. Shahed: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *2015 IEEE 31st international conference on data engineering* (2015), IEEE, pp. 1585–1596. (pg 17)

[66] Erwig, M., and Schneider, M. Developments in spatio-temporal query languages. In *Proceedings. Tenth International Workshop on Database and Expert Systems Applications. DEXA 99* (1999), IEEE, pp. 441–449. (pg 12)

[67] Erwig, M., and Schneider, M. Spatio-temporal predicates. *IEEE Transactions on Knowledge and Data Engineering 14*, 4 (2002), 881–901. (pg 12)

[68] Etemad, M., Júnior, A. S., Hoseyni, A., Rose, J., and Matwin, S. A trajectory segmentation algorithm based on interpolation-based change detection strategies. In *EDBT/ICDT Workshops* (2019), p. 58. (pg 14)

[69] Evagorou, G., Lavalle, M., and Heinis, T. The vantage index: Executing distance queries at scale. In *32nd International Conference on Scientific and Statistical Database Management* (2020), pp. 1–4. (pg 21)

[70] Fanaeepour, M., Kulik, L., Tanin, E., and P Rubinstein, B. I. The case histogram: privacy-aware processing of trajectory data using aggregates. *GeoInformatica 19*, 4 (2015), 747–798. (pp 2, 63, 65, 66, 67, & 88)

[71] Fang, Y., R.Cheng, W.Tang, S.Maniu, and X.Yang. Scalable Algorithms for Nearest-Neighbor Joins on Big Trajectory Data. *IEEE Transactions on Knowledge and Data Engineering 28*, 3 (2016). (pg 40)

[72] Ferragina, P., and Vinciguerra, G. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment (PVLDB) 13*, 8 (2020). (pp 3, 33, 34, 35, 36, 96, 97, 99, 101, & 125)

[73] Finkel, R. A., and Bentley, J. L. Quad trees a data structure for retrieval on composite keys. *Acta informatica 4* (1974), 1–9. (pg 23)

[74] Fox, A., Eichelberger, C., Hughes, J., and Lyon, S. Spatio-temporal indexing in non-relational distributed databases. In *2013 IEEE International Conference on Big Data* (2013), IEEE, pp. 291–299. (pg 16)

[75] Frank, A. U., and Barrera, R. The fieldtree: A data structure for geographic information systems. In *Symposium on Large Spatial Databases* (1989), Springer, pp. 29–44. (pg 22)

[76] Freeman, J. The modelling of spatial relations. *Computer graphics and image processing 4*, 2 (1975), 156–171. (pg 10)

[77] Friedler, S. A., and Mount, D. M. Spatio-temporal range searching over compressed kinetic sensor data. In *European Symposium on Algorithms* (2010), Springer, pp. 386–397. (pg 63)

[78] Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., and Kraska, T. Fiting-tree: A data-aware index structure. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2019), pp. 1189–1206. (pp 33, 34, 35, 99, & 101)

[79] Gao, J., Cao, X., Yao, X., Zhang, G., and Wang, W. Lmsfc: A novel multidimensional index based on learned monotonic space filling curves. *arXiv preprint arXiv:2304.12635* (2023). (pg 4)

[80] Gao, J., and Guibas, L. Geometric algorithms for sensor networks. *Philosophical transactions.Series A, Mathematical, physical, and engineering sciences 370*, 1958 (Jan 13 2012), 27–51. (pg 67)

[81] García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., and Manolopoulos, Y. Efficient distance join query processing in distributed spatial data management systems. *Information Sciences 512* (2020). (pg 41)

[82] Ghane, S., Kulik, L., and Ramamohanarao, K. Publishing spatial histograms under differential privacy. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management* (2018), pp. 1–12. (pp 63, 66, 67, & 88)

[83] Ghosh, A., Ding, J., Sarkar, R., and Gao, J. Differentially private range counting in planar graphs for spatial sensing. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications* (2020), pp. 2233–2242. (pg 75)

[84] Gibbons, P. B. *Data Storage and Indexing in Sensor Networks*. Springer US, Boston, MA, 2009, pp. 635–638. (pg 161)

[85] Golab, L., Garg, S., and Özsu, M. T. On indexing sliding windows over online data streams. In *International Conference on Extending Database Technology* (2004), Springer, pp. 712–729. (pg 97)

[86] Golab, L., and Özsu, M. *Data Stream Management*. Synthesis lectures on data management. Morgan & Claypool Publishers, 2010. (pp 12, 95)

[87] Gonzalez, M. C., Hidalgo, C. A., and Barabasi, A.-L. Understanding individual human mobility patterns. *nature 453*, 7196 (2008), 779–782. (pg 1)

[88] Graefe, G., et al. Modern B-tree techniques. *Foundations and Trends in Databases 3*, 4 (2011), 203–402. (pp 18, 125)

[89] Graefe, G., and Larson, P.-A. B-tree indexes and cpu caches. In *Proceedings 17th International Conference on Data Engineering* (2001), pp. 349–358. (pg 17)

[90] Graefe, G., Linville, A., and Shapiro, L. D. Sort vs. hash revisited. *IEEE Transactions on Knowledge and Data Engineering 6*, 6 (1994), 934–944. (pg 9)

[91] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (1984), pp. 47–57. (pp 17, 23)

[92] gvinciguerra. Pgm-index, 2023. Last Accessed: 2023-12-28. (pp 111, 141)

[93] Hadian, A., and Heinis, T. Interpolation-friendly B-trees: Bridging the gap between algorithmic and learned indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2019). (pg 96)

[94] Hadjieleftheriou, M., Kollios, G., Tsotras, V. J., and Gunopulos, D. Efficient indexing of spatiotemporal objects. In *International conference on extending database technology* (2002), Springer, pp. 251–268. (pg 25)

[95] Hagedorn, S., P.Götze, and K.Sattler. The STARK Framework for Spatio-Temporal Data Analytics on Spark. In *Proceedings ofthe 17th Conference on Database Systems for Business, Technology, and the Web (BTW'17)* (2017). (pp 42, 46)

[96] Hall, D. L., and McMullen, S. A. *Mathematical techniques in multisensor data fusion*. Artech House, 2004. (pg 69)

[97] Han, W.-S., Kim, J., Lee, B. S., Tao, Y., Rantzau, R., and Markl, V. Cost-based predictive spatiotemporal join. *IEEE Transactions on Knowledge and Data Engineering 21*, 2 (2008), 220–233. (pg 16)

[98] He, Z., Cai, Z., Cheng, S., and Wang, X. Approximate aggregation for tracking quantiles and range countings in wireless sensor networks. *Theoretical Computer Science 607* (2015), 381–390. (pg 67)

[99] Hoare, C. A. Quicksort. *The computer journal 5*, 1 (1962), 10–16. (pg 9)

[100] Hu, X., Sintos, S., Gao, J., Agarwal, P. K., and Yang, J. Computing complex temporal join queries efficiently. In *Proceedings of the 2022 International Conference on Management of Data* (2022), pp. 2076–2090. (pg 12)

[101] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (USA, 2010), USENIXATC'10, USENIX Association, p. 11. (pg 160)

[102] Intel. Read time-stamp counter, 1997. Last Accessed: 2023-10-14. (pg 142)

[103] Jackins, C. L., and Tanimoto, S. L. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing 14*, 3 (1980), 249–270. (pp 16, 24)

[104] Jacox, E. H., and Samet, H. Spatial join techniques. *ACM Transactions on Database Systems (TODS) 32*, 1 (2007), 7–es. (pg 10)

[105] Jensen, C. S., Soo, M. D., and Snodgrass, R. T. Unifying temporal data models via a conceptual model. *Information systems 19*, 7 (1994), 513–547. (pg 11)

[106] Kaufmann, M., Manjili, A. A., Vagenas, P., Fischer, P. M., Kossmann, D., Färber, F., and May, N. Timeline index: a unified data structure for processing queries on temporal data in sap hana. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 1173–1184. (pg 11)

[107] Kinsey, L. C. *Topology of surfaces*. Springer Science & Business Media, 1997. (pg 73)

[108] Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., and Neumann, T. SOSD: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019). (pp 33, 96, 111, & 141)

[109] Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., and Neumann, T. RadixSpline: A single-pass learned index. In *Proceedings of the International SIGMOD Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)* (2020). (pp 3, 99)

[110] Kraska, T., Alizadeh, M., Beutel, A., Chi, E. H., Ding, J., Kristo, A., Leclerc, G., Madden, S., Mao, H., and Nathan, V. Sagedb: A learned database system. (pg 29)

[111] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2018), pp. 489–504. (pp 3, 30, 32, & 33)

[112] Krause, A., and Golovin, D. Submodular function maximization. *Tractability 3* (2014), 71–104. (pp 67, 78)

[113] Leis, V., Kemper, A., and Neumann, T. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (2013), pp. 38–49. (pg 20)

[114] Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., and Glance, N. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2007), KDD '07, p. 420–429. (pp 65, 77)

[115] Lettich, F., Orlando, S., Silvestri, C., and Jensen, C. S. Manycore gpu processing of repeated range queries over streams of moving objects observations. *Concurrency and Computation: Practice and Experience 29*, 4 (2017), e3881. (pg 67)

[116] Leutenegger, S. T., Lopez, M. A., and Edgington, J. Str: A simple and efficient algorithm for r-tree packing. In *Proceedings 13th international conference on data engineering* (1997), IEEE, pp. 497–506. (pg 24)

[117] Levandoski, J. J., Lomet, D. B., and Sengupta, S. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (2013), pp. 302–313. (pg 19)

[118] Li, G., Zhou, X., and Cao, L. Ai meets database: Ai4db and db4ai. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 2859–2866. (pg 28)

[119] Li, P., Hua, Y., Jia, J., and Zuo, P. Finedex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment 15*, 2 (2021), 321–334. (pp 34, 36, 97, 125, & 127)

[120] Li, P., Lu, H., Zheng, Q., Yang, L., and Pan, G. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the International Conference on Management of Data* (2020). (pg 4)

[121] Li, R., He, H., Wang, R., Huang, Y., Liu, J., Ruan, S., He, T., Bao, J., and Zheng, Y. Just: Jd urban spatio-temporal data engine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)* (2020), IEEE, pp. 1558–1569. (pg 16)

[122] Li, Y., Chow, C.-Y., Deng, K., Yuan, M., Zeng, J., Zhang, J.-D., Yang, Q., and Zhang, Z.-L. Sampling big trajectory data. In *Proceedings of the 24th ACM*

International on Conference on Information and Knowledge Management (New York, NY, USA, 2015), CIKM '15, Association for Computing Machinery, p. 941–950. (pp 67, 88)

[123] Liu, J., Zhang, F., Lu, L., Qi, C., Guo, X., Deng, D., Li, G., Zhang, H., Zhai, J., Zhang, H., et al. G-learned index: Enabling efficient learned index on gpu. *IEEE Transactions on Parallel and Distributed Systems* (2024). (pg 161)

[124] Llavesh, A., Sirin, U., West, R., and Ailamaki, A. Accelerating B+tree search by using simple machine learning techniques. In *Proceedings of the 1st International VLDB Workshop on Applied AI for Database Systems and Applications (AIDB)* (2019). (pg 99)

[125] Ma, Q., Yang, B., Qian, W., and Zhou, A. Query processing of massive trajectory data based on mapreduce. In *Proceedings of the first international workshop on Cloud data management* (2009), pp. 9–16. (pg 17)

[126] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. Tag: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 131–146. (pg 162)

[127] Mao, Y., Kohler, E., and Morris, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), pp. 183–196. (pg 20)

[128] Marcus, R., Kipf, A., van Renen, A., Stoian, M., Misra, S., Kemper, A., Neumann, T., and Kraska, T. Benchmarking learned indexes. *Proceedings of the VLDB Endowment (PVLDB) 14*, 1 (2020), 1–13. (pp 30, 31, 33, & 96)

[129] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., and Tatbul, N. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019). (pg 29)

[130] Marcus, R., and Papaemmanouil, O. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (2018), pp. 1–4. (pg 29)

[131] Marcus, R., and Papaemmanouil, O. Plan-structured deep neural network models for query performance prediction. *arXiv preprint arXiv:1902.00132* (2019). (pg 29)

[132] Matsuyama, T., Nagao, M., et al. A file organization for geographic information systems based on spatial proximity. *Computer Vision, Graphics, and Image Processing 26*, 3 (1984), 303–318. (pg 22)

[133] Mehr, N., and Horowitz, R. A submodular approach for optimal sensor placement in traffic networks. In *2018 Annual American Control Conference (ACC)* (2018), pp. 6353–6358. (pg 67)

[134] microsoft. Alex, 2022. Last Accessed: 2023-12-28. (pp 111, 141)

[135] Mokbel, M. F., Ghanem, T. M., and Aref, W. G. Spatio-temporal access methods. *IEEE Data Eng. Bull. 26*, 2 (2003), 40–49. (pg 17)

[136] Morton, G. M. A computer oriented geodetic data base and a new technique in file sequencing. (pg 27)

[137] Moti, M. H., Simatis, P., and Papadias, D. Waffle: A workload-aware and query-sensitive framework for disk-based spatial indexing. *Proceedings of the VLDB Endowment 16*, 4 (2022), 670–683. (pg 4)

[138] Nascimento, M. A., and Silva, J. R. Towards historical r-trees. In *Proceedings of the 1998 ACM symposium on Applied Computing* (1998), pp. 235–240. (pp 17, 26)

[139] Nathan, V., Ding, J., Alizadeh, M., and Kraska, T. Learning multi-dimensional indexes. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2020). (pg 3)

[140] Nievergelt, J., Hinterberger, H., and Sevcik, K. C. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS) 9*, 1 (1984), 38–71. (pg 22)

[141] Orenstein, J. A. Multidimensional tries used for associative searching. *Information Processing Letters 14*, 4 (1982), 150–157. (pg 23)

[142] Pavlo, A., Butrovich, M., Joshi, A., Ma, L., Menon, P., Van Aken, D., Lee, L., and Salakhutdinov, R. External vs. internal: an essay on machine learning agents for autonomous database management systems. *IEEE bulletin 42*, 2 (2019). (pg 96)

[143] Pelekis, N., Theodoulidis, B., Kopanakis, I., and Theodoridis, Y. Literature review of spatio-temporal database models. *The Knowledge Engineering Review 19*, 3 (2004), 235–274. (pp 11, 13)

[144] Pfoser, D. Indexing the trajectories of moving objects. *IEEE Data Eng. Bull. 25*, 2 (2002), 3–9. (pp 25, 26)

[145] Pfoser, D., C.Jensen, Y.Theodoridis, and P.Greece. Novel Approaches to the Indexing of Moving Object Trajectories. In *International Conference on Very Large Data Bases* (2001). (pp 3, 39, & 42)

[146] Pfoser, D., Jensen, C. S., Theodoridis, Y., et al. Novel approaches to the indexing of moving object trajectories. In *VLDB* (2000), vol. 2000, Citeseer, pp. 395–406. (pp 16, 25)

[147] Price, C. Table lookup techniques. *ACM Computing Surveys (CSUR) 3*, 2 (1971), 49–64. (pg 9)

[148] Qi, J., Liu, G., Jensen, C. S., and Kulik, L. Effectively learning spatial indices. *Proceedings of the VLDB Endowment (PVLDB) 13*, 12 (2020), 2341–2354. (pg 4)

[149] Qian, C., Yi, C., Cheng, C., Pu, G., Wei, X., and Zhang, H. Geosot-based spatiotemporal index of massive trajectory data. *ISPRS International Journal of Geo-Information 8*, 6 (2019), 284. (pg 16)

[150] Rao, J., and Ross, K. A. Making b+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (2000), pp. 475–486. (pg 19)

[151] Ray, S., A.Brown, N.Koudas, R.Blanco, and A.Goel. Parallel in-memory trajectory-based spatiotemporal topological join. In *International Conference on Big Data* (2015). (pp 40, 41)

[152] Robinson, J. T. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data* (1981), pp. 10–18. (pg 22)

[153] Rosenberg, J. B. Geographical data structures compared: A study of data structures supporting region queries. *IEEE transactions on computer-aided design of integrated circuits and systems 4*, 1 (1985), 53–67. (pg 22)

[154] Rupley Jr, M. L. Introduction to query processing and optimization. *Indiana University at South Bend* (2008). (pg 9)

[155] Samet, H. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR) 16*, 2 (1984), 187–260. (pp 17, 23)

[156] Samet, H., and Webber, R. E. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics (TOG) 4*, 3 (1985), 182–222. (pg 23)

[157] Sarkar, R., and Gao, J. Differential forms for target tracking and aggregate queries in distributed networks. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2010), MobiCom '10, Association for Computing Machinery, p. 377–388. (pp 65, 67, 72, & 88)

[158] Schneider, M. Spatial and spatio-temporal data models and languages., 2009. (pg 12)

[159] Schneider, R., and Kriegel, H.-P. The tr*-tree: A new representation of polygonal objects supporting spatial queries and operations. In *Workshop on Computational Geometry* (1991), Springer, pp. 249–263. (pg 10)

[160] Segev, A., and Shoshani, A. Logical modeling of temporal data. *ACM Sigmod Record 16*, 3 (1987), 454–466. (pg 11)

[161] Sellis, T., Roussopoulos, N., and Faloutsos, C. The r+-tree: A dynamic index for multi-dimensional objects. (pg 24)

[162] Shahvarani, A., and Jacobsen, H.-A. Parallel index-based stream join on a multi-core CPU. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (New York, NY, USA, 2020), Association for Computing Machinery, pp. 2523–2537. (pp 19, 98, 125, & 142)

[163] Shang, S., Chen, L., Z.Wei, C.Jensen, K.Zheng, and P.Kalnis. Parallel trajectory similarity joins in spatial networks. *The VLDB Journal 27*, 3 (2018). (pg 40)

[164] Shang, S., L., K.Zheng, C.Jensen, Z.Wei, and P.Kalnis. Parallel Trajectory-to-Location Join. *IEEE Transactions on Knowledge and Data Engineering 31*, 6 (2019). (pp 40, 41, & 43)

[165] Shang, Z., G.Li, and Z.Bao. Dita: Distributed in-memory trajectory analytics. In *Proceedings of the International Conference on Management of Data* (2018). (pp 39, 42)

[166] Shang, Z., Li, G., and Bao, Z. Dita: distributed in-memory trajectory analytics. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 725–740. (pg 17)

[167] Shi, W., Kong, Q.-J., and Liu, Y. A gps/gis integrated system for urban traffic flow analysis. In *2008 11th International IEEE Conference on Intelligent Transportation Systems* (2008), pp. 844–849. (pg 72)

[168] Shneier, M. Two hierarchical linear feature representations: edge pyramids and edge quadtrees. *Computer Graphics and Image Processing 17*, 3 (1981), 211–224. (pg 23)

[169] Šidlauskas, D., Šaltenis, S., Christiansen, C. W., Johansen, J. M., and Šaulys, D. Trees or grids? indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL international conference on Advances in Geographic Information Systems* (2009), pp. 236–245. (pg 16)

[170] Silberschatz, A., Korth, H. F., and Sudarshan, S. Database system concepts. (pg 8)

[171] Sriharsha, R. Magellan: Geospatial analytics on spark, 2015. (pg 41)

[172] Su, H., Liu, S., Zheng, B., Zhou, X., and Zheng, K. A survey of trajectory distance measures and performance evaluation. *The VLDB Journal 29* (2020), 3–32. (pg 14)

[173] Sun, J., and Li, G. An end-to-end learning-based cost estimator. *Proc. VLDB Endow. 13*, 3 (nov 2019), 307–319. (pg 29)

[174] Sun, Z., Zhou, X., and Li, G. Learned index: A comprehensive experimental evaluation. *Proceedings of the VLDB Endowment 16*, 8 (2023), 1992–2004. (pg 33)

[175] Ta, N., G.Li, Y.Xie, C.Li, S.Hao, and J.Feng. Signature-based trajectory similarity join. *IEEE Transactions on Knowledge and Data Engineering 29*, 4 (2017). (pg 40)

[176] Tamminen, M. Encoding pixel trees. *Computer Vision, Graphics, and Image Processing 28*, 1 (1984), 44–57. (pg 22)

[177] Tampakis, P., Doulkeridis, C., Pelekis, N., and Y.Theodoridis. Distributed Subtrajectory Join on Massive Datasets. *ACM Transactions Spatial Algorithms Systems 6*, 2 (2020). (pg 40)

[178] Tang, C. Finedex, 2022. Last Accessed: 2023-12-28. (pg 141)

[179] Tang, C. Xindex, 2022. Last Accessed: 2023-12-28. (pg 141)

[180] Tang, C., Wang, Y., Dong, Z., Hu, G., Wang, Z., Wang, M., and Chen, H. Xindex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2020), pp. 308–320. (pp 34, 35, 36, 97, 125, & 127)

[181] Tang, M., Yu, Y., Malluhi, Q., Ouzzani, M., and Aref, W. LocationSpark: A Distributed in-Memory Data Management System for Big Spatial Data. *VLDB 9*, 13 (2016). (pp 39, 41, & 46)

[182] Tao, Y., Kollios, G., Considine, J., Li, F., and Papadias, D. Spatio-temporal aggregation using sketches. In *Proceedings. 20th International Conference on Data Engineering* (2004), IEEE, pp. 214–225. (pp 1, 63, 65, 66, & 67)

[183] Tao, Y., and Papadias, D. Efficient historical r-trees. In *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001* (2001), IEEE, pp. 223–232. (pp 17, 26)

[184] Tao, Y., and Papadias, D. The mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of Very Large Data Bases Conference (VLDB), 11-14 September, Rome* (2001). (pp 17, 26)

[185] Tao, Y., and Papadias, D. Historical spatio-temporal aggregation. *ACM Transactions on Information Systems (TOIS) 23*, 1 (2005), 61–102. (pp 1, 63)

[186] Theodoridis, Y., Vazirgiannis, M., and Sellis, T. Spatio-temporal indexing for large multimedia applications. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems* (1996), IEEE, pp. 441–448. (pp 16, 24)

[187] Theodoropoulos, G. S., Nørvåg, K., and Doulkeridis, C. Efficient semantic similarity search over spatio-textual data. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2024). (pg 161)

[188] Tian, R., Zhai, H., Zhang, W., Wang, F., and Guan, Y. A survey of spatio-temporal big data indexing methods in distributed environment. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 15* (2022), 4132–4155. (pg 28)

[189] van Oosterom, P. Spatial access methods. *Geographical information systems 1* (1999), 385–400. (pp 21, 22, & 27)

[190] Wang, C., Yu, J., and Zhao, Z. Glin: A (g) eneric (l) earned (in) dexing mechanism for complex geometries. In *Proceedings of the 11th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data* (2023), pp. 1–12. (pp 3, 37)

[191] Wang, H., and Zimmermann, R. Processing of continuous location-based range queries on moving objects in road networks. *IEEE Transactions on Knowledge and Data Engineering 23*, 7 (2011), 1065–1078. (pg 66)

[192] Wang, L., Zheng, Y., Xie, X., and Ma, W.-Y. A flexible spatio-temporal indexing scheme for large-scale gps track retrieval. In *The Ninth International Conference on Mobile Data Management (mdm 2008)* (2008), IEEE, pp. 1–8. (pp 17, 27)

[193] Wang, S., Bao, Z., Culpepper, J. S., and Cong, G. A survey on trajectory data management, analytics, and learning. *ACM Computing Surveys (CSUR) 54*, 2 (2021), 1–36. (pg 14)

[194] Wang, Y., Zheng, Y., and Xue, Y. Travel time estimation of a path using sparse trajectories. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (2014), pp. 25–34. (pg 1)

[195] Whitman, R., B.Marsh, M.Park, and E.Hoel. Distributed spatial and spatio-temporal join on apache spark. *ACM Transactions Spatial Algorithms Systems 5*, 1 (2019). (pp 17, 42, 46, 50, & 55)

[196] Winther, M., Christensen, J. H., Plejdrup, M. S., Ravn, E. S., Eriksson, Ó. F., and Kristensen, H. O. Emission inventories for ships in the arctic based on satellite sampled ais data. *Atmospheric Environment 91* (2014), 1–14. (pg 1)

[197] Woltmann, L., Hartmann, C., Thiele, M., Habich, D., and Lehner, W. Cardinality estimation with local deep learning models. In *Proceedings of the second international workshop on exploiting artificial intelligence techniques for data management* (2019), pp. 1–8. (pg 29)

[198] Wu, J., Zhang, Y., Chen, S., Wang, J., Chen, Y., and Xing, C. Updatable learned index with precise positions. *arXiv preprint arXiv:2104.05520* (2021). (pp 33, 34)

[199] Xia, Y., and Prabhakar, S. Q+ rtree: Efficient indexing for moving object databases. In *Eighth International Conference on Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings.* (2003), IEEE, pp. 175–182. (pg 67)

[200] Xie, D., F.Li, B.Yao, G.Li, L.Zhou, and M.Guo. Simba: Efficient in-memory spatial analytics. In *International Conference on Management of Data* (2016). (pp 39, 42)

[201] Xu, X., Han, J., and Lu, W. Rt-tree: An improved r-tree indexing structure for temporal spatial databases [c]. In *The International Symposium on Spatial Data Handling (SDH). Zurich* (1990), pp. 1040–1049. (pp 17, 26)

[202] Yeong-Sung, F., Cheng-Ta, and Hsu, Y.-Y. An energy-efficient algorithm for object tracking in wireless sensor networks. In *2010 IEEE International Conference on Wireless Communications, Networking and Information Security* (2010), pp. 424–430. (pp 70, 74)

[203] Yeong-Sung, F., Hsu, Y.-Y., et al. An energy-efficient algorithm for object tracking in wireless sensor networks. In *2010 IEEE international conference on wireless communications, networking and information security* (2010), IEEE, pp. 424–430. (pg 68)

[204] Ying, J. J.-C., Lee, W.-C., Weng, T.-C., and Tseng, V. S. Semantic trajectory mining for location prediction. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems* (2011), pp. 34–43. (pg 161)

[205] You, S., Zhang, J., and L.Gruenwald. Large-scale spatial join query processing in cloud. In *International Conference on Data Engineering Workshops* (2015). (pp 39, 41, & 46)

[206] Yu, J., Zhang, Z., and Sarwat, M. Spatial data management in apache spark: the geospark perspective and beyond. *GeoInformatica* (2018). (pp 39, 41, & 46)

[207] Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X., Sun, G., and Huang, Y. T-drive: Driving directions based on taxi trajectories. ACM SIGSPATIAL GIS 2010. (pg 88)

[208] Y.Zheng. Trajectory data mining: An overview. *ACM Trans.Intell.Syst.Technol. 6*, 3 (may 2015). (pg 1)

[209] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud)* (2010), USENIX Association, p. 10. (pp 25, 39)

[210] Zhang, C., Li, F., and Jestes, J. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the 15th international conference on extending database technology* (2012), pp. 38–49. (pg 10)

[211] Zhang, H., Lim, H., Leis, V., Andersen, D. G., Kaminsky, M., Keeton, K., and Pavlo, A. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2018), pp. 323–336. (pg 20)

[212] Zhang, J., Liu, Y., Zhou, K., Li, G., Xiao, Z., Cheng, B., Xing, J., Wang, Y., Cheng, T., Liu, L., et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 international conference on management of data* (2019), pp. 415–432. (pp 29, 30)

[213] Zhang, S., J.Han, Z.Liu, K.Wang, and Z.Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *International Conference on Cluster Computing and Workshops* (2009). (pg 41)

[214] Zhang, S., Mao, Y., He, J., Grulich, P. M., Zeuch, S., He, B., Ma, R. T., and Markl, V. Parallelizing intra-window join on multicores: An experimental study. In *Proceedings of the 2021 International Conference on Management of Data* (2021). (pp 111, 141, & 142)

[215] Zhang, Z., Chu, Z., Jin, P., Luo, Y., Xie, X., Wan, S., Luo, Y., Wu, X., Zou, P., Zheng, C., et al. Plin: a persistent learned index for non-volatile memory with high performance and instant recovery. *Proceedings of the VLDB Endowment 16*, 2 (2022), 243–255. (pg 161)

[216] Zhang, Z., C.Jin, J.Mao, X.Yang, and A.Zhou. Trajspark: A scalable and efficient in-memory management system for big trajectory data. In *Web and Big Data* (2017). (pp 39, 42, & 46)

[217] Zheng, Y., Xie, X., and Ma, W.-Y. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data(base) Engineering Bulletin* (June 2010). (pg 88)

[218] Zhong, W., Chen, C., Wu, X., and Jiang, S. REMIX: Efficient range query for LSM-trees. In *USENIX Conference on File and Storage Technologies (FAST)* (2021). (pg 111)

[219] Zhou, P., Zhang, D., Salzberg, B., Cooperman, G., and Kollios, G. Close pair queries in moving object databases. In *Proceedings of the 13th annual ACM international workshop on Geographic information systems* (2005), pp. 2–11. (pp 17, 27)

[220] Zhou, X., Chai, C., Li, G., and Sun, J. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering 34*, 3 (2020), 1096–1116. (pg 28)

[221] Zhu, R., Weng, L., Wei, W., Wu, D., Peng, J., Wang, Y., Ding, B., Lian, D., Zheng, B., and Zhou, J. Pilotscope: Steering databases with machine learning drivers. (pg 29)

[222] Zou, B., You, J., Wang, Q., Wen, X., and Jia, L. Survey on learnable databases: A machine learning perspective. *Big Data Research 27* (2022), 100304. (pg 28)