Predictive In-memory Multi-Level Indexing Algorithm for Spatiotemporal Trajectory Streams in Distributed Environments

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Nam Phung

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Eleazar Leal

June 2024

© Nam Phung 2024

Acknowledgements

In the acknowledgements section of my academic thesis, I extend my heartfelt gratitude to Dr. Eleazar Leal, my advisor, whose unwavering support and invaluable mentorship have been instrumental in bringing this work to fruition. Dr. Leal generously dedicated his time, despite his busy schedule, to guide me through various facets of my academic and professional pursuits.

I am equally thankful to all my teachers for the opportunities they provided and the encouragement that propelled my learning and accomplishments.

Additionally, I am fortunate and grateful to have received dear wishes and motivational support from my sister and parents, who have been the pillars of strength throughout this endeavor.

Abstract

Trajectory analysis has received significant contributions in recent years. With the rapid explosion of GPS-enabled devices, several large-scale datasets have been created, e.g., the Geolife GPS dataset (17,621 trajectories) and the bdd100k dataset (100,000 trajectories). This has provided enormous streaming spatiotemporal data, benefiting many real-world applications, e.g., urban planning, mapping services, and carpooling. These applications benefit from performing many types of search queries on spatial data, such as range query and join query. Despite the importance of these types of queries on streaming data, many systems do not support them. Many also fail to handle the scalability and efficiency problems when the input data is too large. This thesis proposes the first predictive in-memory multi-level indexing algorithm called PIMMLI. We introduced predictive indexing to enhance the scalability of the indexing process and compared it against an existing state-of-the-art algorithm called DITA. We have conducted extensive experiments on real-world streaming datasets and compared the performance of PIMMLI against DITA on different hyperparameters. Our results show that PIMMLI (1) has a similar range query performance with DITA; (2) has at least a 5.8% improvement in join query performance compared to DITA; (3) has an average improvement of 28.10% for trajectory indexing.

Contents

Contents iii

List of Tables vi

List of Figures vii

1 Introduction 1

1.1 Background 1

1.1.1 Trajectory 2

1.1.2 Trajectory Stream 3

1.1.3 Trajectory Distance 3

1.1.4 Range Query 4

1.1.5 Join Query 5

1.1.6 Spatial Data Indexing 6

1.1.7 Distributed Computing 8

1.2 Problem Statement 9

1.3 Challenges 10

1.4 Related Works 11

1.5 Contributions 13

2 Algorithm Description 15
2.1 Introduction . . . 15
2.2 Batch Processing . . . 16
2.3 Predictive Indexing . . . 17
2.3.1 Model Training . . . 17
2.3.2 Model Validation . . . 18
2.3.3 Point Prediction . . . 19
2.4 Multi-Level Indexing . . . 23
2.4.1 Global Index . . . 23
2.4.2 Local Index . . . 27

3 Experimental Results 30
3.1 Experiments . . . 30
3.1.1 Experimental Setup . . . 30
3.1.2 Hardware . . . 31
3.1.3 Datasets . . . 32
3.1.4 Parameters . . . 33
3.1.5 Evaluation Metrics . . . 35
3.2 Results . . . 35
3.2.1 Impact of Global Index Branching Factor N_G . . . 35
3.2.2 Impact of Local Partitions N_L . . . 41
3.2.3 Impact of Sampling Rate r . . . 46
3.2.4 Impact of Spark Workers N_w . . . 51

4 Conclusions & Future Work 56
4.1 Conclusions . . . 56
4.2 Future Work . . . 59

References **61**

List of Tables

3.1 Hardware configurations 31

3.2 PIMMLI parameters 33

3.3 Apache Spark parameters 33

List of Figures

1.1 Example of a 2-dimensional spatiotemporal trajectory 2
1.2 Example of a range query 5
1.3 Example of a join query 6
1.4 Examples of valid and invalid MBRs 7
1.5 Spatial data points and MBRs in an R-tree 7
1.6 R-Tree representation of figure 1.5 8
1.7 Spark master-slave architecture 9

2.1 Continuous Data Stream Discretization. 17
2.2 Example of distance d and angle θ between two spatial points 1 and 2 18
2.3 Example of batch training 19
2.4 Example of batch prediction 21
2.5 Example of spatial partitioning 24
2.6 Example of global index on start-points of the trajectories 25
2.7 Example of global index on end points of the trajectories 26
2.8 Pivot point selection using the example in figure 2.5. 28
2.9 Local indexes of the example in figure 2.5 29

3.1 Visualization of sampled GeoLife GPS trajectory dataset 32
3.2 Cumulative Range Query Time of PIMMLI when Varying N_G 36

3.3 Cumulative Range Query Time of PIMMLI and DITA when Varying N_G . . . 37
3.4 Cumulative Join Query Time of PIMMLI when Varying N_G 38
3.5 Cumulative Join Query Time of PIMMLI and DITA when Varying N_G 39
3.6 Cumulative Index Time of PIMMLI when Varying N_G 40
3.7 Cumulative Index Time of PIMMLI and DITA when Varying N_G . . 40
3.8 Cumulative Range Query Time of PIMMLI when Varying N_L 41
3.9 Cumulative Range Query Time of PIMMLI and DITA when Varying N_L . . . 42
3.10 Cumulative Join Query Time of PIMMLI when Varying N_L 43
3.11 Cumulative Join Query Time of PIMMLI and DITA when Varying N_L 44
3.12 Cumulative Index Time of PIMMLI when Varying N_L 45
3.13 Cumulative Index Time of PIMMLI and DITA when Varying N_L . . 45
3.14 Cumulative Range Query Time of PIMMLI when Varying r 46
3.15 Cumulative Range Query Time of PIMMLI and DITA when Varying r 47
3.16 Cumulative Join Query Time of PIMMLI when Varying r 48
3.17 Cumulative Join Query Time of PIMMLI and DITA when Varying r 49
3.18 Cumulative Index Time of PIMMLI when Varying r 50
3.19 Cumulative Index Time of PIMMLI and DITA when Varying r . . . 50
3.20 Cumulative Range Query Time of PIMMLI when Varying N_w 51
3.21 Cumulative Range Query Time of PIMMLI and DITA when Varying N_w . . . 52
3.22 Cumulative Join Query Time of PIMMLI when Varying N_w 53
3.23 Cumulative Join Query Time of PIMMLI and DITA when Varying N_w 53
3.24 Cumulative Index Time of PIMMLI when Varying N_w 54
3.25 Cumulative Index Time of PIMMLI and DITA when Varying N_w . . 55

1 Introduction

1.1 Background

This section presents the formal definitions of terms used in trajectory analysis algorithms together with existing challenges, related works on trajectory analysis, and our contributions to this field of research. Trajectory analysis involves analyzing and processing the movement patterns of objects in various applications. These algorithms aim to extract meaningful information from trajectory data, such as identifying patterns, predicting future movements, or detecting anomalies. Trajectory analysis can benefit many real-world applications such as transportation optimization, navigation systems, bioinformatics, and road network planning [1, 2, 3, 4, 5]. However, it is a challenge to process and mine trajectory data [6]. Firstly, efficiently storing a large amount of trajectory data on a single machine is extremely difficult. Secondly, indexing and querying trajectory data pose significant challenges in terms of time and space complexity. Addressing these challenges requires a combination of domain knowledge, algorithmic advancements, and computational resources. In this research, we propose PIMMLI, a predictive trajectory analysis algorithm designed to overcome these challenges.

1.1.1 Trajectory

The increasing use of GPS-enabled devices has created a tremendous amount of trajectory data capturing movements of human and vehicles. Trajectory data is a type of data that can be found in a spatial-temporal dataset. This data can be generated by technologies such as Global Position Systems (GPS), RFID sensors, and GSM beacons [7]. Trajectory is defined as a sequence of points generated from a moving object and is ordered by time. This data can be generated by any devices that incorporate GPS technology. A single point in a trajectory contains spatial-attributes such as longitude and latitude. Several other non-spatial-attributes such as time, id, device specification can also be included in a single data point of a trajectory.

By definition, a point p is a location in d-dimensional space, i.e. $p \in \mathbb{R}^d$. A trajectory τ is a sequence of points from a moving object ordered by time, i.e., $\tau = (p_0, p_1, p_2, ..., p_n | p_i \in \mathbb{R}^d)$. In this paper, we assume the objects traverse a two-dimensional space, i.e., $p_i \in \mathbb{R}^2$. The trajectory data points also include other attributes such as timestamp, user ID, name, and home city [8].

Figure 1.1 shows an example of a 2-dimensional trajectory of an object in which each point p_i represents the object's location at timestamp t_i. Therefore, the trajectory in this figure can be formally defined as $\tau = ((p_i, t_i) | p_i \in \mathbb{R}^2, t_i \in \mathbb{R}^+)$.

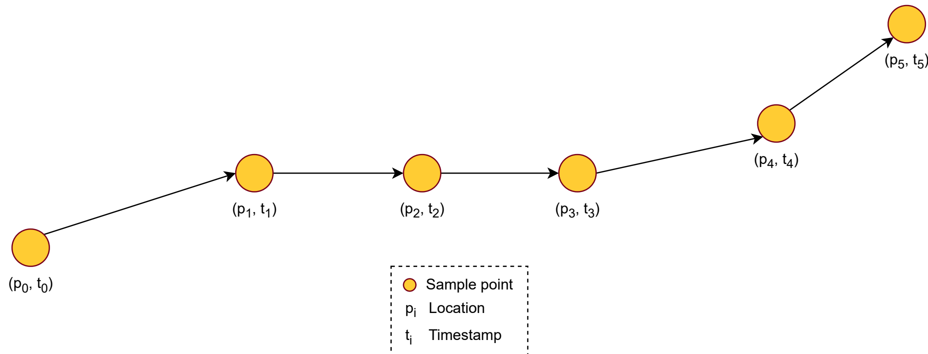

Figure 1.1: Example of a 2-dimensional spatiotemporal trajectory

1.1.2 Trajectory Stream

A trajectory stream is a series of points p_i ordered by time that records the location of a moving object at timestamp i. A trajectory stream is an unbounded stream of time-series data [9, 10]. Therefore, it does not have a start and end point. A streaming trajectory algorithm must be able to respond to multiple real-time queries on the streaming data [11]. In order to achieve this, a sliding window of size w is applied to the data. The window slides forward as more streaming data arrive and captures at most w data points in the trajectories. In this research, we use the notion of the sliding window to discretize the trajectory stream into separate batches and perform data indexing and querying on them.

1.1.3 Trajectory Distance

A trajectory distance measure is a function that returns the distance between two trajectories. We denoted the function as D and the distance between two trajectories τ_1 and τ_2 as $D(\tau_1, \tau_2)$. There are several trajectory distance measures such as Fréchet distance, EDR, LCSS, DTW, DFD, LSED, etc [12]. Dynamic Time Warping (DTW) [13]) is a robust and popular method in trajectory analysis. DTW is a two-level DP-matching algorithm [14] that finds the best alignment between a reference pattern and another pattern [15]. This is equivalent to minimizing the Euclidean distance between two aligned spatial-temporal trajectories. Therefore, we will use DTW as the similarity measure for PIMMLI.

Given two trajectories $\tau_1 = (x_0, x_1, x_2, ..., x_m)$ and $\tau_2 = (y_0, y_1, y_2, ..., y_n)$, the

DTW between them is defined as follows

$$DTW(\tau_1, \tau_2) = \begin{cases} \sum_{i=0}^{m} dist(x_i, y_0), & \text{if } n = 0 \\ \sum_{j=0}^{n} dist(x_0, y_j), & \text{if } m = 0 \\ dist(x_m, y_n) + min(DTW(\tau_1^{m-1}, \tau_2^{n-1}), & \\ DTW(\tau_1^{m-1}, \tau_2), DTW(\tau_1, \tau_2^{n-1})), & \text{otherwise,} \end{cases} \tag{1.1}$$

where τ_1^{m-k} is the trajectory τ_1 without the last k points and $dist(x_m, y_n)$ is the Euclidean distance between the data points x_m and y_n

$$dist(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} \tag{1.2}$$

1.1.4 Range Query

Given a point p in a two-dimensional space, i.e. $p \in \mathbb{R}^2$, a spatial range threshold $\epsilon \in \mathbb{R}^+$, a trajectory-based distance measure d, and a set T of trajectories, the range query operation finds all trajectories $\tau \in T$ such that $D(p, p_i) < \epsilon, \exists p_i \in \tau$. Range queries find all trajectories that intersect or fall into the spatial range ϵ [16, 17]. For example, range queries can assist road planning by querying all vehicles that pass through a specific road in the city between a time interval. This data can then be used to determine the traffic flow and road development [18]. Figure 1.2 shows an example of a range query that retrieves all trajectories that interest with a circle centered at point p with radius ϵ.

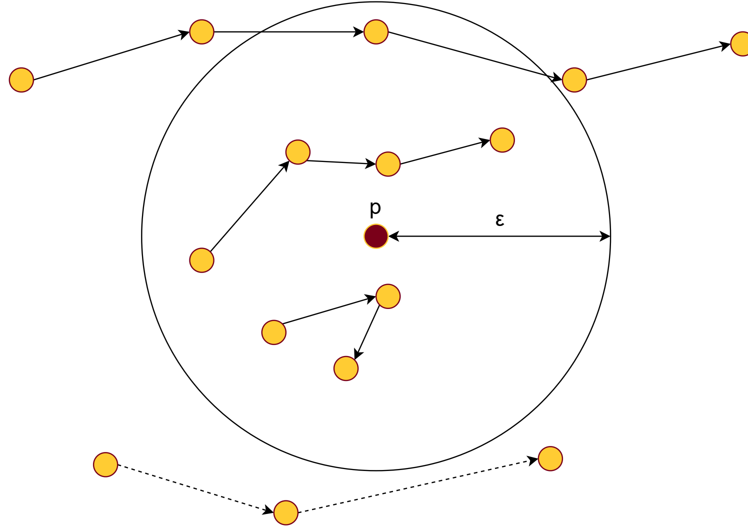

Figure 1.2: Example of a range query

1.1.5 Join Query

Join query retrieves information from two or more spatial data sets [19]. An example of a join query is: "Find all vehicles that use the same road from Los Angeles to Oregon". This information could be visualized and provide insight into the problem of interstate highway development [20]. Given sets of trajectories T_1 and T_2, a trajectory distance threshold ϵ, a trajectory-based distance measure D, the join query operation finds all pairs of trajectories (τ_i, τ_j) from the Cartesian product $T_1 \times T_2$ such that $D(\tau_i, \tau_j) < \epsilon$. Figure 1.3 shows an example of a join query that finds trajectories with similar shape from two different data sets.

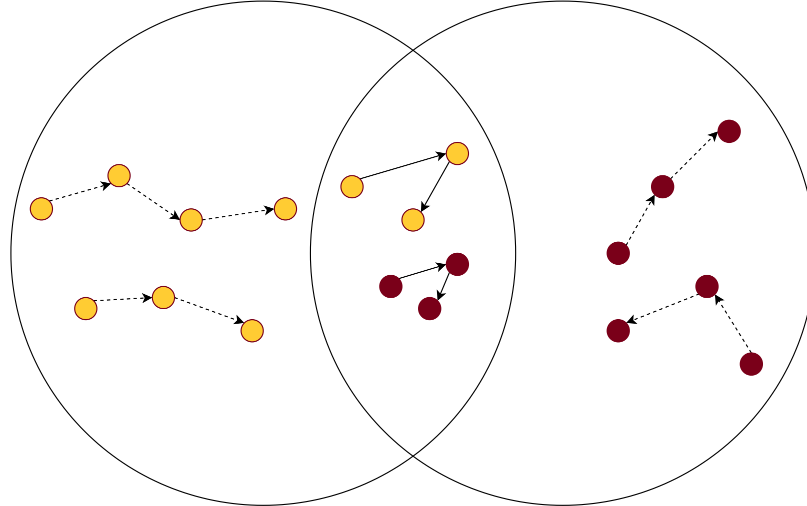

Figure 1.3: Example of a join query

1.1.6 Spatial Data Indexing

The fast retrieval of large trajectory data is crucial for trajectory analysis algorithms. These algorithms rely on a specialized data structure that can effectively capture the spatial characteristics of the data. One such data structure is the R-Tree, which has gained recognition and is widely used in spatial database management systems [21, 22, 23, 24, 25]. The R-Tree is a height-balanced tree data structure that efficiently stores spatial data. Each node in the R-Tree represents a region with minimum coverage area that encompasses all of its child nodes, known as the minimum bounding rectangle (MBR). Figure 1.4 provides an example of a valid MBR that covers all the spatial data points and two other invalid MBRs in which one fails to capture all data points and the other does not have a minimum coverage area.

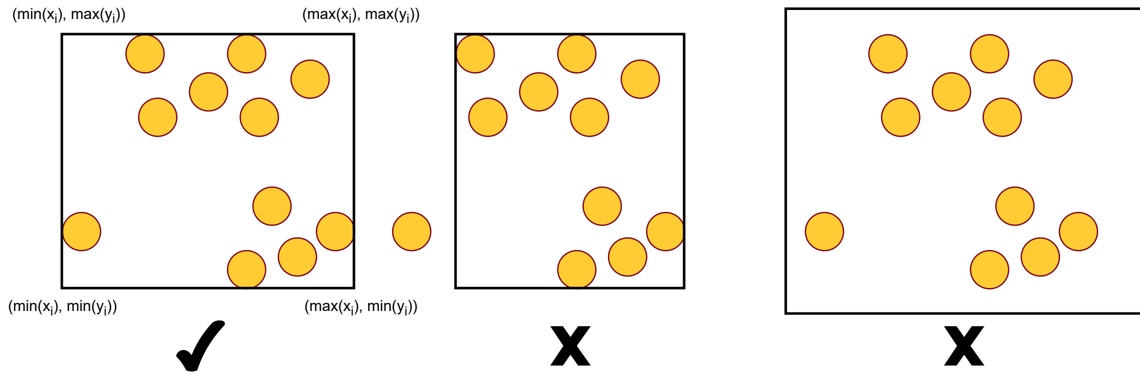

Figure 1.4: Examples of valid and invalid MBRs

The leaf nodes contain the actual spatial data points in the database. The search key of an R-tree is the MBR of each node. Querying data in an R-tree takes as input a circle-shaped query region described by its center point and its radius [26]. Figure 1.5 shows an example of how the data are distributed spatially and figure 1.6 shows the tree representation of the data.

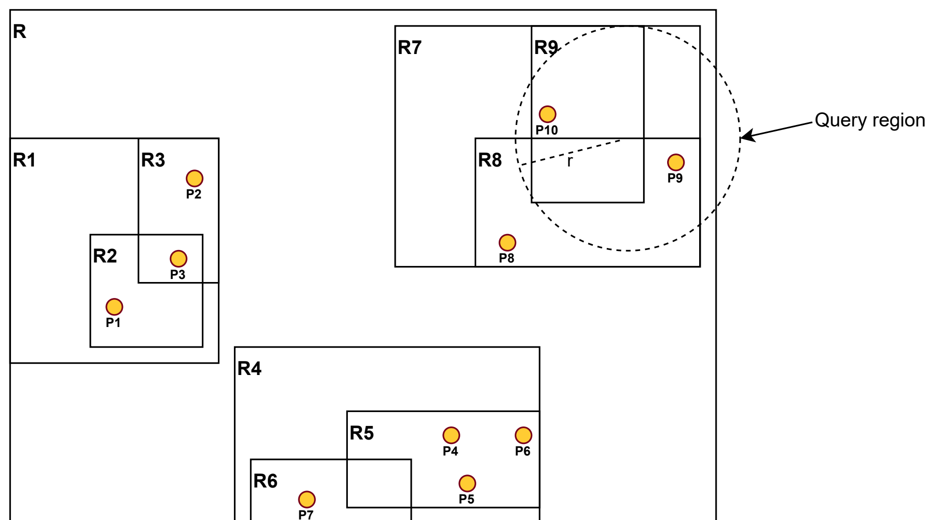

Figure 1.5: Spatial data points and MBRs in an R-tree

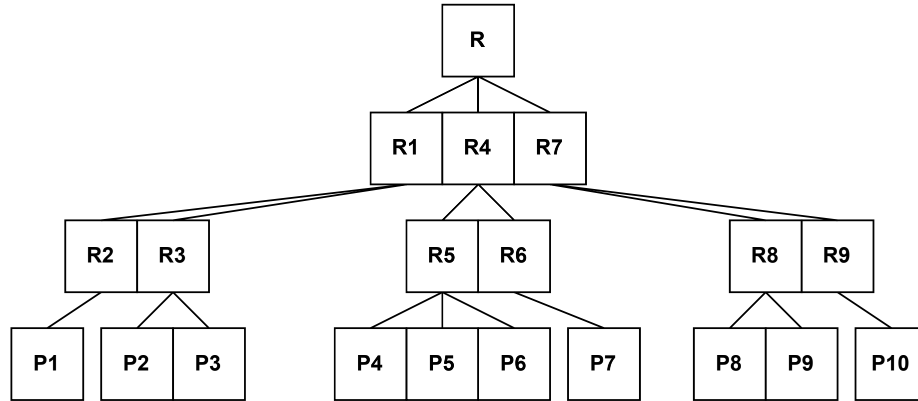

Figure 1.6: R-Tree representation of figure 1.5

1.1.7 Distributed Computing

Due to the increasing amount of trajectory data, centralized systems for processing data pose a great efficiency challenge. To tackle this problem, many state-of-the-art trajectory analysis algorithms use distributed computing systems [27, 28, 29]. These systems allow data partitioning and parallel computation across multiple nodes within a distributed network. Data partitioning is an operation in distributed systems to divide data evenly across different network nodes to ensure load-balancing [30]. This allows the spatial query time to be divided among the nodes, thus, reducing the total execution time of a query. Moreover, combined with filtering techniques, the query will not need to access the nodes that do not contain the candidate results, which can further reduce the execution time and transfer costs.

Apache Spark

Apache Spark is an engine for processing big data processing on clusters. It has been adopted as a reliable and scalable framework by the spatial database development community [31, 32]. Figure 1.7 shows the master-slave architecture of a Spark system. The driver runs the master node and the executors run across the worker

nodes in the cluster. A Spark cluster manager is used to acquire cluster resources for executing Spark jobs. The cluster manager handles resource sharing between Spark nodes [33]. The stream data arrives at the driver and the cluster manager partition the data across the worker nodes. The executors process the spatial queries and send the results on worker nodes back to the driver program that resides on the master node.

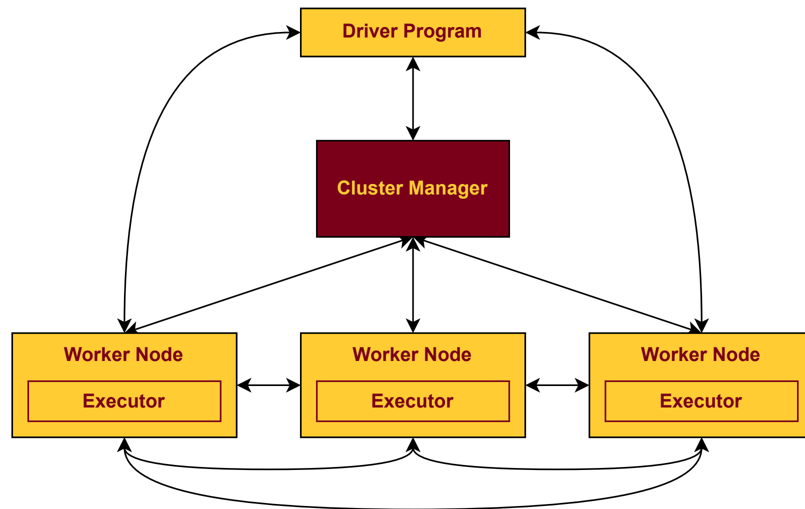

Figure 1.7: Spark master-slave architecture

1.2 Problem Statement

Trajectory analysis algorithm suffers from a problem of high cost in time and space in data indexing and querying operation. Indexing trajectory data requires large data storage with high scalability, which often presents a tremendous problem in centralized systems [34]. In addition, many algorithms do not support parallel search in important query operations such as range query and join query [35, 36, 37]. Previous work on PIMMLI focused solely on developing the algorithm that works on streaming data in centralized systems. This poses a challenge when dealing

with large-scale streaming trajectory data. In this work, we touch on the problem of distributed computing, enabling PIMMLI to perform in distributed environments and benchmark its performance against other state-of-the-art distributed trajectory analysis algorithm.

1.3 Challenges

Several challenges have been identified in the field of trajectory analysis and are the main focus of many research on trajectory analysis [38, 39, 40]. Below, we provide the discussion on some known challenges within this field.

Data Imbalanced

The efficient storage and fast retrieval of large data is important in trajectory analysis. One aspect of the data storage requires trajectory algorithms to handle data balancing efficiently. This requires using data structures suitable for spatial indexing such as R-Tree [27, 41], Quad-tree [42], Grid-index [35], and Frame structure [43]. Furthermore, trajectories represent spatial streaming data characterized by irregular distribution and increasing data size over time. Therefore, an efficient data partitioning technique [44, 45, 46] can enhance the performance by balancing the data across multiple partitions. Since these partitions are disjoint, the trajectory algorithm can perform trajectory data mining in parallel using a distributed computing framework such as Spark. However, many algorithms suffer from performance issues due to lacking support for distributed computing [35, 36, 37, 47, 48].

Similarity Analysis

Many similarity measures have been used to compare two trajectories. Measurements that performs trajectory-to-trajectory comparison include Fréchet, DTW, ERP, Hausdorff, LCSS, and EDwP [49, 50, 13, 51, 52, 53, 54]. Many similarity measures also suffer from time complexity issue and noise in data [55]. Moreover, computing distances between two trajectories generally requires a dynamic programming solution with a time complexity of $\mathcal{O}(N^2)$.

1.4 Related Works

Numerous studies have been conducted on the analysis of trajectory data. Additionally, many of these studies primarily focus on offline trajectory data, which may not be suitable for real-world applications that involve streaming data in an online fashion. Below, we provide an overview of several existing works in this field, along with their respective findings.

Simba

Simba [56], also known as Spatial In-Memory Big data Analytics, provides a scalable and efficient solution for spatial query and analysis. By extending the Spark SQL engine, Simba supports streaming trajectory data and various spatial queries such as range query and kNN query. It incorporates a query optimizer that ensures low latency and high throughput in spatial data indexing. Experimental results have demonstrated that Simba outperforms other spatial data analytics systems. For instance, Simba achieves 51 times lower latency and 45 times higher throughput than SpatialHadoop for kNN queries. Additionally, when compared to Spark SQL, Simba's performance remains largely unaffected by the size of the dataset.

DITA

The DITA [41] algorithm introduces a data partitioning mechanism that addresses the issue of data locality by utilizing global and local indexes. These indexes are built using an R-tree structure to index the pivotal spatial points of trajectories, enabling trajectory pruning and enhancing query efficiency. DITA also incorporates a filter-verification framework that estimates trajectory similarity using pivot points, allowing for the removal of dissimilar pairs. DITA supports both join query and range query on streaming data. In benchmark tests against the Simba [56] and DFT [57], DITA demonstrated superior scalability. This is attributed to its load-balancing mechanism, resulting in a higher scale-up performance. Additionally, DITA exhibits greater scale-out capability, enabling the handling of larger datasets compared to Simba.

UlTraMan

UlTraMan [58] is a comprehensive platform that leverages Apache Spark for managing and analyzing large trajectory data. UlTraMan supports ID query, range query, kNN query, and trajectory clustering on streaming data. It incorporates a unified storage and computing engine, integrating a key-value store within Spark's internal block manager. This integration enables efficient data processing and reliable data management. UlTraMan enhances Spark's MapReduce and RDD capabilities to create TrajDataset, a distributed computing paradigm that efficiently handles distributed computational operations at global and local levels. Comparative results demonstrate that UlTraMan outperforms baseline methods in terms of performance for both range and join queries.

Dragoon

Dragoon [27] trajectory management system is based on Spark. It supports ID query, range query, and kNN query by featuring a mutable resilient distributed dataset (mRDD), including RDD Share, RDD Update, and RDD Mirror, which enables hybrid storage of non-streaming and streaming trajectories. This is essentially different from other existing algorithms that use Spark immutable RDD. Experimental results showed that Dragoon has a similar offline trajectory query performance compared to the UlTraMan [58] algorithm. Dragoon decreases up to doubled storage overhead compared with UlTraMan during trajectory editing. Dragoon improves scalability by at least 40% compared to other streaming processing frameworks such as Spark Streaming. It also doubles the performance for online trajectory data analysis.

1.5 Contributions

This thesis introduces a highly efficient trajectory search algorithm, specifically designed for use in distributed environments. We present a novel algorithm called Predictive In-Memory Multi-Level Indexing (PIMMLI) that addresses the challenge in data indexing. We build on previous thesis research of Abdul Samad [59] by incorporating the join query operation, addressing current bugs, implementing the brute force algorithm, deploying PIMMLI in a distributed environment, and evaluating its performance against benchmark algorithms. Our main research contribution lies in incorporating predictive indexing into trajectory analysis, with the aim of reducing the indexing time. This is pivotal for the development of spatial databases capable of processing real-time large-scale spatio-temporal data. Our experiments were conducted using 9 computers from the computer science lab in conjunction with UMD's Ukko server. We evaluated the performance of PIMMLI against DITA by varying different

hyperparameters that govern spatial data indexing and querying. The experimental results demonstrate that PIMMLI outperforms DITA by 28.10% in terms of indexing time and exhibits similar performance in range query and 5.8% improvement in join query. In Chapter 2, we outline the proposed algorithm and provide details on the predictive indexing and multi-level indexing processes. Chapter 3 presents the findings from our investigations, while Chapter 4 concludes the thesis and discusses future work.

2 Algorithm Description

This chapter explains the main idea behind PIMMLI. We first present the discretization of continuous data stream for indexing purpose in 2.2. We then discuss and provide the pseudo-code for predictive indexing in 2.3. This involves the algorithm for predicting future spatial points and model validation for retraining models. In 2.4, we present the implementation of multi-level indexing mechanism, which is used to select candidate partitions as well as candidate trajectories for querying similar spatial trajectories.

2.1 Introduction

PIMMLI utilizes Apache Spark for processing large-scale data. It handles streaming spatial trajectory data by applying a fixed-length window to the data stream to create separate batches. Subsequently, it conducts spatial point prediction for each trajectory within the batch. The prediction algorithm utilizes the existing data points from previous batches to predict the next set of spatial points for each trajectory. In the event of an incorrect prediction by the trajectory model, a new model is trained and updated. The discussion and pseudo-code for spatial data discretization and predictive indexing can be found in section 2.2 and section 2.3, respectively.

To create both the global and local index, the trajectories are indexed using the R-tree data structure. PIMMLI utilizes global partitioning to separate the trajectories into distinct partitions and distributes them across nodes in the distributed network.

This global partitioning ensures that similar trajectories are stored within the same partition by indexing their first and last spatial points. This optimization enhances the query process by filtering out trajectories that do not share the same starting and ending spatial points. Additionally, the local index partitions the trajectories within each global partition based on their pivot points. As a result, the local index can group trajectories with similar shapes together, facilitating more efficient query operations. In section 2.4, we present an in-depth discussion of multi-level indexing and the pseudo-code of the global and local indexing process.

2.2 Batch Processing

The increasing prevalence of GPS devices leads to a significant rise in the volume of spatial data being generated. This poses a major challenge for trajectory analysis algorithms, as the data size is unbounded. When the data becomes larger, many algorithms struggle to perform real-time analysis and indexing efficiently. To address this issue, PIMMLI employs a data discretization technique by processing incoming data points in separate batches. PIMMLI periodically reads the data and ensures that each batch has the same size. Subsequently, PIMMLI conducts model training and prediction on one batch at a time, using the data points to create the database index. Figure 2.1 provides an illustration of continuous spatial data batch processing.

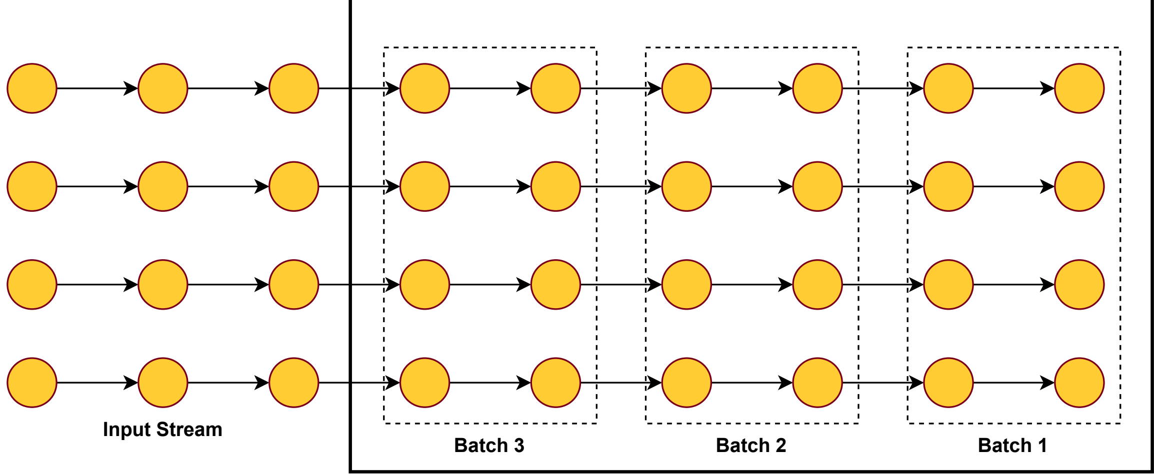

Figure 2.1: Continuous Data Stream Discretization.

2.3 Predictive Indexing

2.3.1 Model Training

Per each trajectory, PIMMLI constructs a predictive model to predict forthcoming points along that trajectory based on its most recent w points p_n, p_{n-1}, p_{n-2}, ..., p_{n-w+1}. The model predicts the subsequent k points within the trajectory by computing the average Euclidean distance $\bar{d}$ and the average angle $\bar{\theta}$ computed between successive pairs of points at any time instance t. We present the formula for computing the average distance and average angle as follows

$$\bar{d}(\tau, w) = \frac{\sum_{i=n-w+1}^{n-1} dist(p_i, p_{i+1})}{w-1} \tag{2.1}$$

$$\bar{\theta}(\tau, w) = \frac{\sum_{i=n-w+1}^{n-1} \arctan\left(\frac{p_{i+1}.y - p_i.y}{p_{i+1}.x - p_i.x}\right)}{w-1}, \tag{2.2}$$

where $\tau = (p_1, p_2, ..., p_n)$ and $dist(p_i, p_j)$ is the function that computes the Euclidean distance between two spatial points p_i and p_j of τ. Figure 2.2 illustrates an example of the distance d and angle θ of two points in a trajectory.

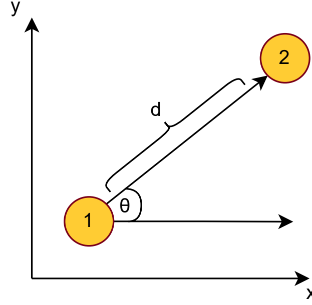

Figure 2.2: Example of distance d and angle θ between two spatial points 1 and 2

One notable benefit associated with the utilization of this model lies in its simplicity. Given that PIMMLI needs to efficiently handle substantial volumes of trajectories in real-time, it becomes imperative that the algorithm updates all models and generate predictions in a timely manner.

2.3.2 Model Validation

PIMMLI leverages predicted points to achieve a one-step advantage in the indexing phase. It expands the minimum bounded rectangles (MBRs) to initiate the periodic reconstruction of the R-tree, as opposed to performing this operation during each batch processing iteration. Before predicting the future points of a trajectory, we validate the model by computing the average Euclidean distance and angle of the latest w points. We then compute the average distance $\bar{d}_T$ and angle $\bar{\theta}_T$ among all trajectories. Finally, we compute the Euclidean distance of $(\bar{d}_T, \bar{\theta}_T)$ against the model's $(\bar{d}_M, \bar{\theta}_M)$. If the distance exceeds a threshold ϵ, we retrain all trajectory models and rebuild the indexes.

2.3.3 Point Prediction

We apply a training window of size w on every batch of data to train the predictive model of the trajectories. At each batch prediction, we also predict the k future points for each trajectory. The predicted points will be prepended to the trajectories for indexing purposes. We use the spatial points to build the local and global index of the trajectories.

Figure 2.3 illustrates an example of applying a training window of size w on a batch data at time i to predict the future points of each trajectory. The streaming data is discretized by applying a fixed-size window to create separate batches of data. When PIMMLI receives the first batch, it applies a fixed-size training window on the w most-recent data points and train the models for the trajectories in the batch. The result models make prediction on the incoming data points of the trajectories and temporarily prepend the predicted points to the trajectories. When PIMMLI processes the next batch, it removes the previous predicted points from the trajectories and prepend the new data in the batch to the existing trajectories. It then repeats the entire process again.

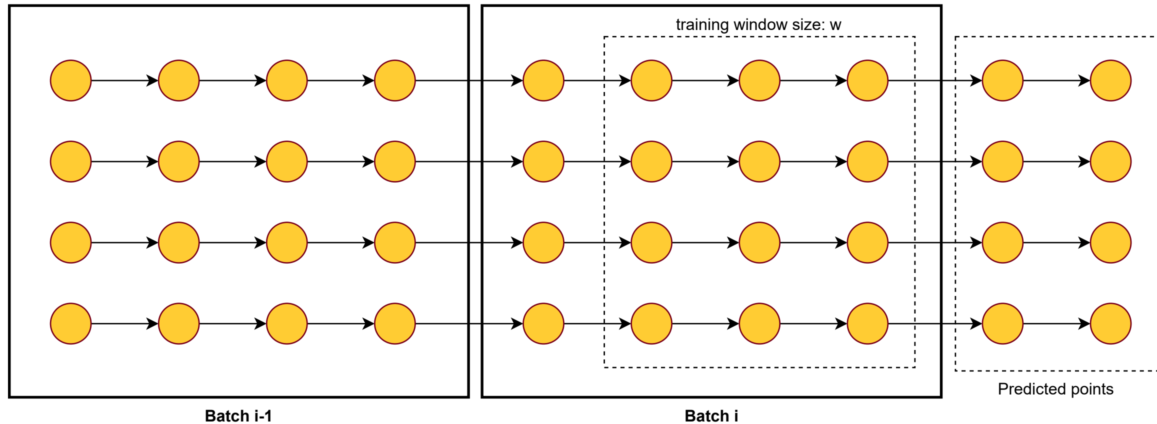

Figure 2.3: Example of batch training

Each trajectory has its own predictive model. We use the model to predict the

future points of the trajectory based on the average distance and angle of each consecutive pair of spatial points. The future point can be predicted by displacing the last point p_n in the trajectory τ by a distance $\bar{d}$ with an angle $\bar{\theta}$. The formula for computing the coordinate of the next point is $p_{n+1} = (p_{n+1}.x, p_{n+1}.y)$ where

$$\begin{aligned} p_{n+1}.x &= p_n.x + \bar{d} \cdot \cos \bar{\theta} \\ p_{n+1}.y &= p_n.y + \bar{d} \cdot \sin \bar{\theta}. \end{aligned} \tag{2.3}$$

We present an example of the future point prediction and the pseudo-code for predictive indexing, model training, model validation, and point prediction in figure 2.4, algorithm 1, 2, 3, and 4 below respectively.

In the example in figure 2.4, the trajectory τ has four data points $\tau = (p_1, p_2, p_3, p_4)$. To predict the next future point, we use the coordinates of the points $p_1, ..., p_4$. We compute the Euclidean distance and angle in radian of the successive pairs of points by using these formulas

$$d(p_i, p_{i-1}) = \sqrt{(p_i.x - p_{i-1}.x)^2 + (p_i.y - p_{i-1}.y)^2} \tag{2.4}$$

$$\theta(p_i, p_{i-1}) = \arctan\left(\frac{p_i.y - p_{i-1}.y}{p_i.x - p_{i-1}.x}\right) \tag{2.5}$$

This gives us $(d_{1,2}, \theta_{1,2}) = (2, 0)$, $(d_{2,3}, \theta_{2,3}) = (2.5, 0.643)$, $(d_{3,4}, \theta_{3,4}) = (2.66, 0.345)$. We compute the displacement of the predicted point from the last point in the trajectory by finding the average distance $\bar{d} = \frac{2+2.5+2.66}{3} = 2.39$ and angle $\bar{\theta} = \frac{0+0.643+0.345}{3} = 0.33$. We then compute the coordinate of the predicted point using the formula 2.3. The predicted point of the trajectory τ has a coordinate $p_{pred} = (8.76, 3.17)$.

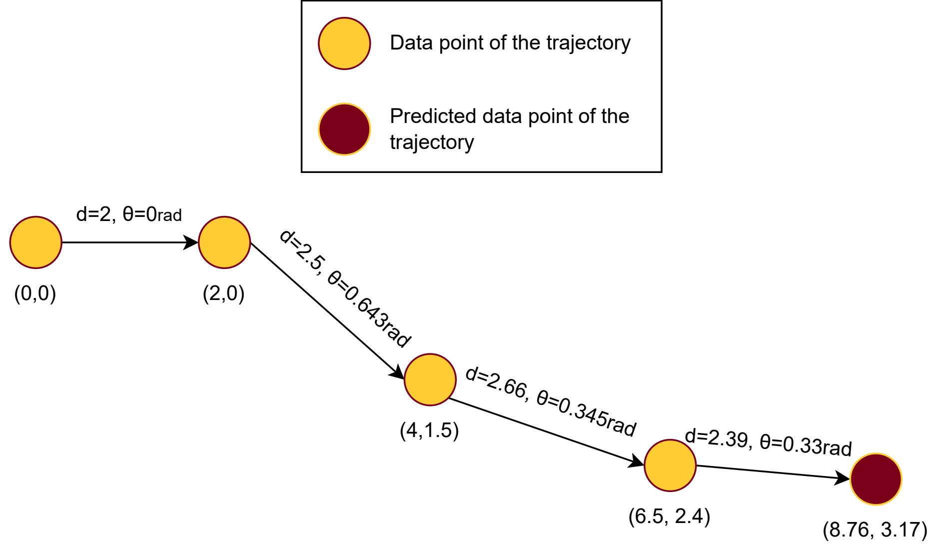

Figure 2.4: Example of batch prediction

Algorithm 1 Predictive Indexing

Require: db: trajectory database, $models$: predictive models, $batch$: current batch, i: current batch index, t: training interval, w: training window size, k: prediction size, ϵ: validation threshold

1: **function** PROCESSBATCH(db, $models$, $batch$, i, t, w, k, ϵ)
2:     $db \leftarrow db \cup batch$
3:     **if** ($i = 1$ or $i \bmod t = 0$) or (VALIDATE(db, $models$, $batch$, ϵ)$= False$) **then**
4:         $models \leftarrow$ TRAIN(db, $models$, w)
5:         $extendedDB \leftarrow db$
6:         **for all** $trajectory$ in db **do**
7:             $points \leftarrow$ PREDICT($models$, $trajectory$, k)
8:             $extendedDB[trajectory] \leftarrow trajectory$.CONCAT($points$)
9:         **end for**
10:         $indexes \leftarrow$ BUILDINDEXES($extendedDB$)
11:     **end if**
12: **end function**

Algorithm 2 Model Training

```
1: function TRAIN(db, models, w)
2:     for all trajectory in db do
3:         n ← |trajectory| − w
4:         trainSet ← trajectory[n : n + w + 1]
5:         models[trajectory] ← DISTANDANGLE(trainSet)
6:     end for
7:     return models
8: end function
```

Algorithm 3 Model Validation

```
1: function VALIDATE(db, models, batch, ϵ)
2:     errors ← {}
3:     for all trajectory in db do
4:         curHeadings ← DISTANDANGLE(models[trajectory])
5:         nxtHeadings ← DISTANDANGLE(batch[trajectory])
6:         errors[trajectory] ← DIST(curHeadings, nxtHeadings)
7:     end for
8:     return MEAN(errors) < ϵ
9: end function
```

Algorithm 4 Point Prediction

```
1: function PREDICT(models, trajectory, k)
2:     lastPoint ← trajectory[last]
3:     futurePoints ← []
4:     for i ← 1 to k do
5:         dist, angle ← DISTANDANGLE(models[trajectory])
6:         nextPoint ← MOVE(lastPoint, dist, angle)
7:         futurePoints.APPEND(nextPoint)
8:         lastPoint ← futurePoints[last]
9:     end for
10:    return futurePoints
11: end function
```

2.4 Multi-Level Indexing

The data points are indexed using a multi-level indexing scheme. This scheme utilizes an R-Tree structure to distribute the trajectories across the nodes in the distributed network. The global index is responsible for grouping trajectories with similar start and end points, as defined by their minimum bounding rectangles (MBRs), into the same partitions. On the other hand, the local index is utilized to divide the trajectories within each global partition into multiple local partitions based on their pivot points. As a result, trajectories within the same local partition will share similarities in terms of their starting points, ending points, and shapes.

2.4.1 Global Index

The process of indexing trajectories begins by extracting the first and last points of each trajectory. We utilize these points to create global partitions that group the trajectories based on their respective first and last points. We group the start points into distinct buckets. Within each bucket, we further group the trajectories based on their end points, creating multiple sub-buckets. Each sub-bucket forms a partition. These partitions are then stored on the nodes across the network.

Figure 2.5 illustrates an example of forming the global partitions of a spatial dataset. In the example, the trajectories are grouped into multiple MBRs by their first points, forming three rectangles that contain the trajectory $\{F_1, F_2, F_5, F_7\}$, $\{F_3, F_6\}$, $\{F_4\}$. For each MBR, we group the trajectories inside by their last points and form the sub MBRs. For example, the MBR $\{F_1, F_2, F_5, F_7\}$ has 2 sub MBRs $\{L_1\}$ and $\{L_2, L_5, L_7\}$. This procedure groups the trajectories into multiple partitions, forming a tree-like structure in the figure below where each leaf node is a partition containing the actual trajectory.

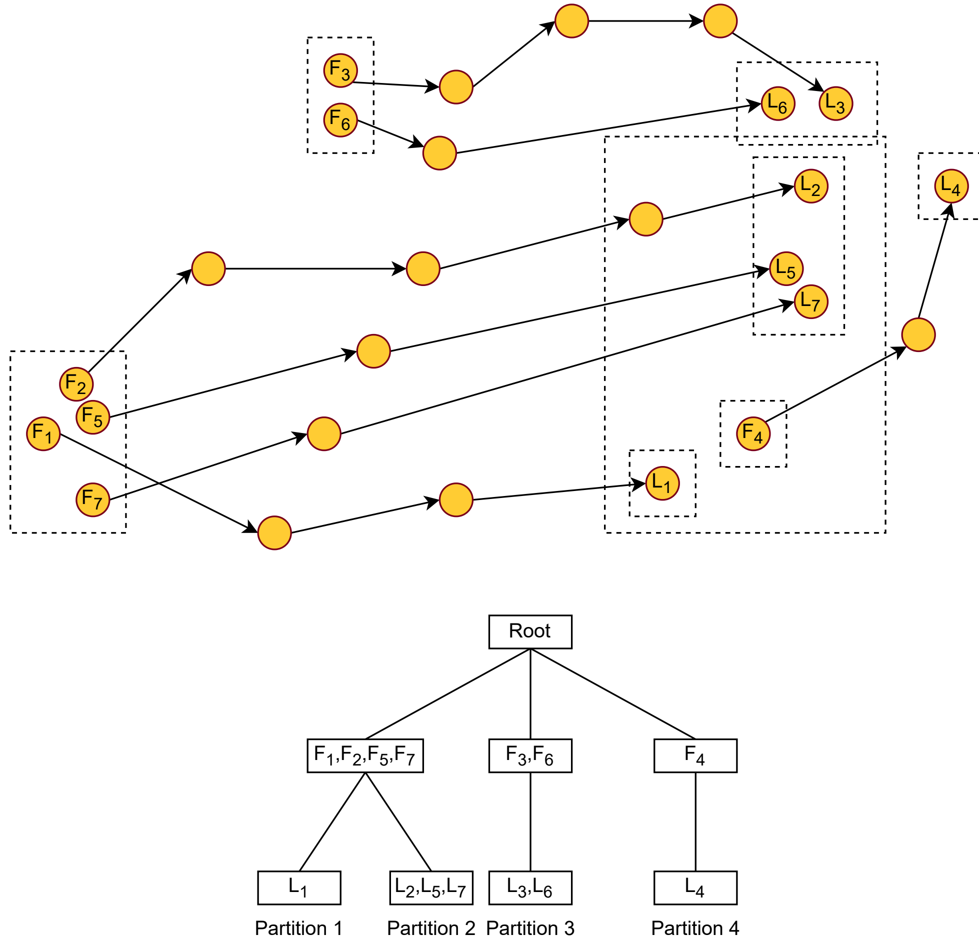

Figure 2.5: Example of spatial partitioning

Subsequently, we use the data points to construct two R-Trees for trajectories indexing through their start points and end points. The collection of all start points forms a global minimum bounding rectangle (MBR), which is the root node of the first R-Tree in figure 2.6. The points within the global MBR are grouped together to create sub-MBRs recursively. Each sub-MBR forms an internal node within the first R-Tree. Additionally, trajectories with closely located end points form a second global MBR. Similar steps are then followed to generate the second indexing R-Tree for these end points. The leaf nodes of the trees contain references to the partitions holding the trajectories. By utilizing the two R-Trees, we can efficiently locate the

candidate trajectories with starting and ending points that are in proximity to the input query.

Figure 2.6 and 2.7 show the example of constructing the start-points and end-points R-Tree index from the example in figure 2.5, respectively. We also present the pseudo-code for constructing the global index in algorithm 5 below. Figure 2.6 shows all first points of the trajectories in figure 2.5. We start by constructing the rectangle that captures all data points, forming the global MBR. For all points in the global MBR, we group them by their distance, forming the sub MBRs which correspond to the sub rectangles A, F, and G in the figure below. For each sub MBR, we then repeat the steps above recursively. Finally, this constructs the global index on the start points of the trajectories, which is a R-Tree structure where each node is a MBR and the leaf nodes reference to the partitions holding the actual data.

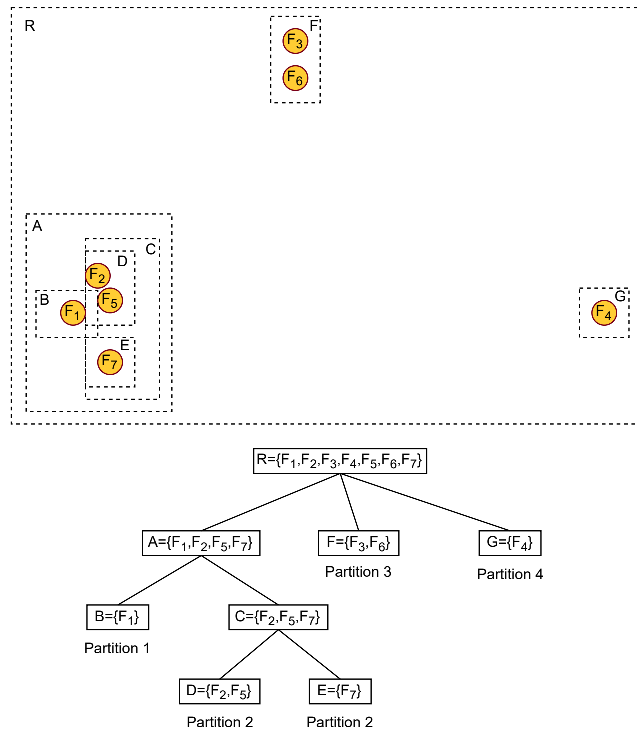

Figure 2.6: Example of global index on start-points of the trajectories

In figure 2.7, we repeat the same steps using the end points $\{L_1, L_2, .., L_7\}$ of the trajectories. The R-Tree created from this procedure allows us to find the trajectories that have the end points in close proximity to the end point of the input query. Using the two R-Trees above, we can use a bidirectional search to quickly locate partitions and the candidate trajectories.

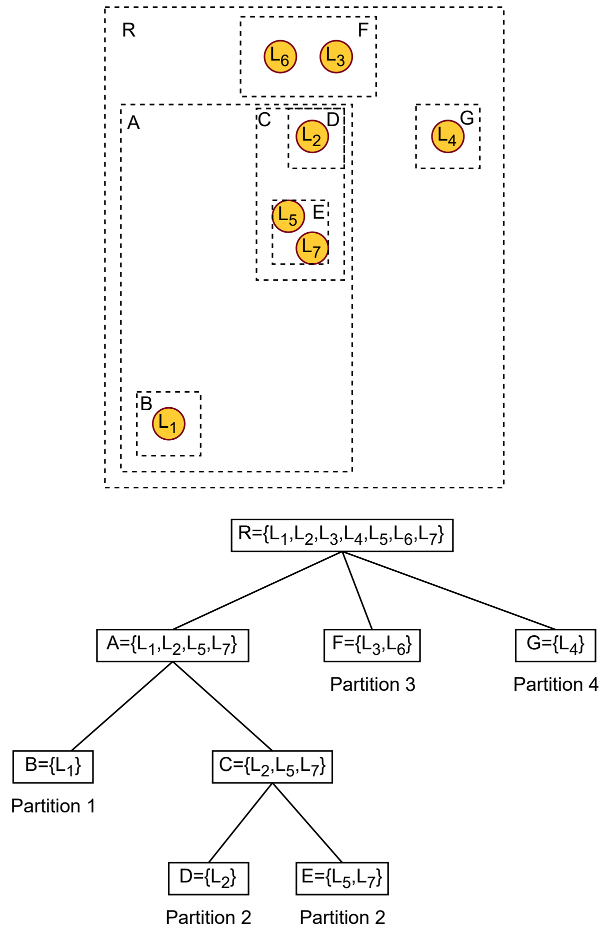

Figure 2.7: Example of global index on end points of the trajectories

Algorithm 5 Global Index Construction

Require: *db*: trajectory database

```
1:  function buildGlobalIndex(db)
2:      Group trajectories in db into multiple buckets based on the first points
3:      for each bucket do
4:          Group trajectories in bucket into multiple sub-buckets based on the last points
5:          The trajectories in each sub-bucket form a partition
6:      end for
7:      Build R-Tree T_first using the first points
8:      Build R-Tree T_last using the last points
9:      return partitions, T_first, T_last
10: end function
```

2.4.2 Local Index

We create a tree-like structure for each global partition to index the trajectories within it. To construct the root MBR that covers all points, we choose pivot points from the trajectories along with their start points and end points. In the second level of the R-Tree, we group the start points into sub-MBRs, which become the children of the root node. The third level of the tree consists of MBRs that cover the end points of the trajectories.

The subsequent levels of the tree consist of MBRs that cover the pivot points $1, 2, ..., n$ of the trajectories. A pivot point is defined as a point of inflection. Given a trajectory τ, for any three consecutive data points A, B, C, we measure the angle formed by these points $\measuredangle ABC$ in radians. The value $\pi - \measuredangle ABC$ is the weight for point B. A large weight indicates that B is a pivot point of τ. Figure 2.9 provides an illustration of constructing local indexes, while algorithm 6 below presents the corresponding pseudo-code.

In this example figure 2.8, we locate the pivot points of each trajectory. For example, trajectory 1 has only 1 pivot point with a maroon color. The angle formed

by the point F_1, the pivot point, and the point after it is greater than a certain threshold such that if we remove the pivot point, the shape of the trajectory will be affected. Therefore, we use it in the local indexing step. Trajectory F_1 also contains a non-pivot point. Since the angle formed by the non-pivot point, the point before it, and the point L_1 is relatively straight, therefore, removing it will not affect the shape of the trajectory. We continue to identify the pivot points of the remaining trajectories using the point of inflection strategy.

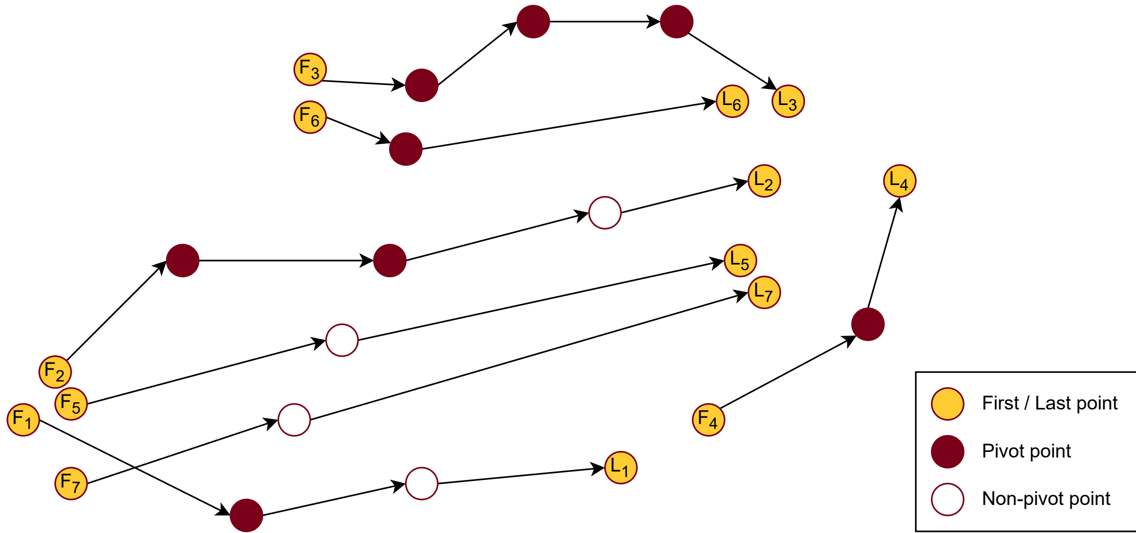

Figure 2.8: Pivot point selection using the example in figure 2.5.

After identifying the pivot points, for each partition, we construct a local index based on the pivot points of the trajectories stored in that partition. The local index is a tree-like structure where the second level of the tree contains the first points of the trajectories, the third level contains the last points, and each level after that contains the ith pivot point of the trajectories. Similar to the global indexing step, each node in the local index tree forms an MBR that groups the neighboring points together. The constructed local indexes from figure 2.8 are illustrated in the figure 2.9 below. Using the local indexes, we can narrow down the list of candidate trajectories

by filtering out the trajectories that do not share similar shape.

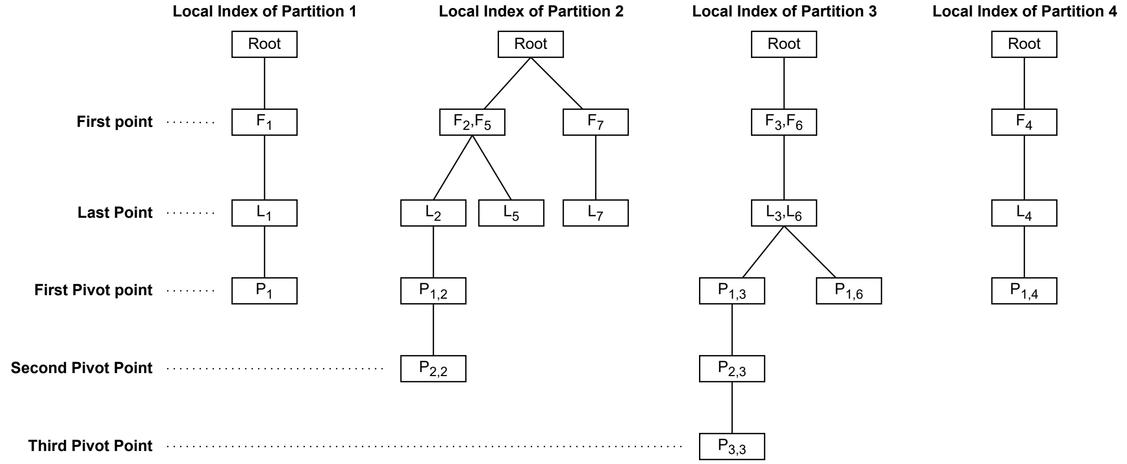

Figure 2.9: Local indexes of the example in figure 2.5

Algorithm 6 Local Index Construction

Require: *partitions*: all partitions storing the trajectories

1: **function** BUILDLOCALINDEX(*partitions*)
2:     **for all** *pt* in *partitions* **do**
3:         Create *root* node containing all *trajectories* in *pt*
4:         *trajs* ← FILTERFIRSTLASTPIVOT(*trajectories*)
5:         LOCALINDEX(*trajs*, *root*)
6:     **end for**
7: **end function**
8:
9: **function** LOCALINDEX(*trajectories*, *root*)
10:     *p* ← *trajectories*.HEAD
11:     Group *trajectories* into multiple buckets based on *p*
12:     **for all** *bucket* in *buckets* **do**
13:         Build a *node* for *bucket* using the MBR constructed from *p*
14:         *root*.ADDCHILD(*node*)
15:         *trajs* ← *trajectories*.TAIL
16:         LOCALINDEX(*trajs*, *node*)
17:     **end for**
18: **end function**

3 Experimental Results

This chapter examines the experimental outcomes and performance evaluation of PIMMLI. Section 3.1 presents the experimental setup, hardware specifications, dataset, parameters setup, and evaluation metrics in our studies. Section 3.2 delves into our discoveries and assesses the performance of PIMMLI when varying different parameters in a distributed environment.

3.1 Experiments

3.1.1 Experimental Setup

We compiled PIMMLI using Scala 2.11.12, Java 1.8.0, Maven 3.6.2 and ran the algorithm using Apache Spark 2.2.0. Spark Engine provides high-level APIs for data engineering in distributed computing. We also setup a Python script to emulate the data streaming process. This script generates streaming trajectory files, which are then stored in a directory monitored by PIMMLI. Each trajectory file comprises a timestamp, trajectory ID, and trajectory points separated by forward slashes. The trajectory points are delimited by semicolons, and each data point contains coordinates separated by commas. Below is an illustration of a trajectory data file:

```
1    0/38c1c0ab176d4b5f612c42c7/79.95,232.66;79.99,232.61;79.94,232.65
2    0/241e648b5c44750a7b184afe/79.91,232.83;79.91,232.83;79.91,232.83
```

We ran the experiments on the Ukko server at the University of Minnesota-Duluth.

In addition to that, we also utilized the machines from the Computer Science lab e.g. csdev02, csdev03, csdev05, etc. as the worker nodes for our distributed computing network. Section 3.1.2 presents in detail the specifications of these machines.

3.1.2 Hardware

We used the machines provided by the Computer Science Department of the University of Minnesota Duluth. This includes the Ukko server and 9 other machines, i.e. csdev02, csdev03, csdev05, etc. All machines use Ubuntu version 22.04.3 LTS (Jammy Jellyfish). The hardware configurations for all machines are listed in Table 3.1 below.

Name	**CPU**	**RAM**
ukko	Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz 56 threads (28 cores, 2 threads/core, 2 sockets)	512GB
csdev05	13th Gen Intel(R) Core(TM) i9-13900KF 48 threads (24 cores, 2 threads/core, 1 socket)	32GB
csdev06	12th Gen Intel(R) Core(TM) i9-12900K 32 threads (16 cores, 2 threads/core, 1 socket)	
csdev02	11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz 16 threads (8 cores, 2 threads/core, 1 socket)	16GB
csdev03		
csdev07		
csdev08		
csdev09		
csdev10		
csdev11		

Table 3.1: Hardware configurations

3.1.3 Datasets

The Geolife GPS trajectory dataset provided by Microsoft Research Asia includes GPS data collected from 182 users over a period of more than three years [60, 61, 62]. This dataset captures users' outdoor activities such as commuting, sports, shopping, dining, and more. The trajectories were recorded using various GPS devices with different sampling rates. The dataset comprises 17,621 trajectories covering a total distance of 750,000 miles, 24.8 million data points, and an average trajectory length of 1,411 data points. These trajectories were recorded in Beijing, China and are depicted in Figure 3.1.

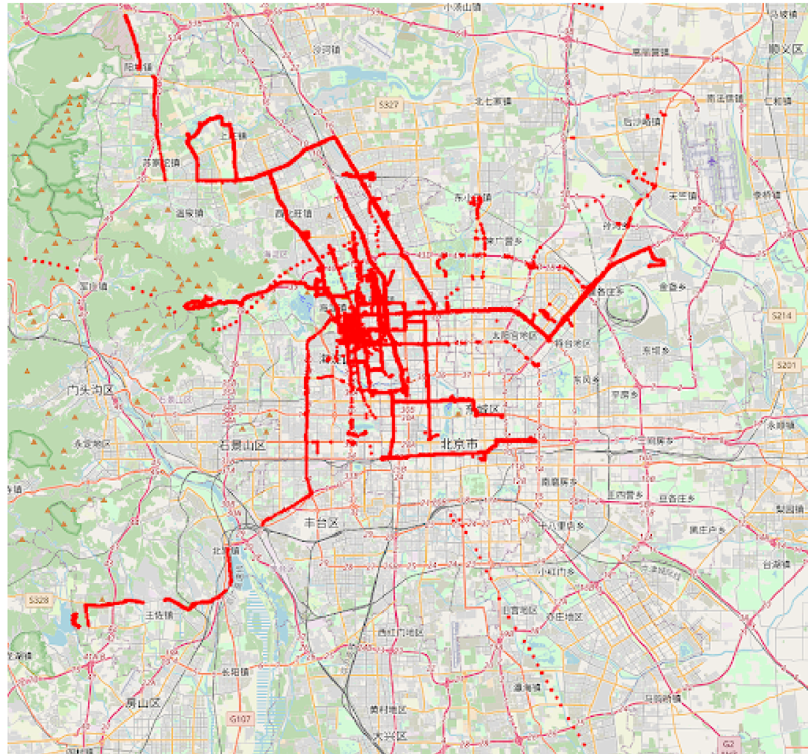

Figure 3.1: Visualization of sampled GeoLife GPS trajectory dataset

3.1.4 Parameters

Table 3.2 and 3.3 list the parameters used in our experiments. The default value of all parameters are highlighted in bold. When we varied a parameter, the other parameters were set to their default values. We present further descriptions of these parameters below.

Parameter	Value	Description
N_G	4, 8, **16**, 32, 64	Branching factor of global index
N_L	4, **8**, 16, 32, 64	Branching factor of local index
r	**0.25**, 0.5, 0.75, 1	Global index MBRs sampling rate

Table 3.2: PIMMLI parameters

Parameter	Value	Description
N_w	**3**, 6, 9	Number of workers
W_m	**16**	Worker memory (GB)

Table 3.3: Apache Spark parameters

Global Index Branching Factor N_G

The global index branching factor N_G dictates the number of global partitions. As the global indexing procedure takes in only the first and last points of each trajectory, the height of the index tree remains constant at 2. Each internal node will possess a branching factor of N_G, meaning that every node will have N_G child nodes. Thus, the space complexity of the global index is $\mathcal{O}(N_G * N_G)$. A global index tree with $N_G = 8$ will have at most 64 global partitions.

Local Index Branching Factor N_L

The branching factor N_L determines the number of local partitions that PIMMLI creates for each global partition. Since the local indexing process takes in the first, last, and pivot points of any given trajectory, the number of local partitions depends on the height of the index tree and its branching factor N_L. An index tree with height h and branching factor N_L will have a space complexity of $\mathcal{O}(N_L^h)$.

R-Tree Sampling Rate r

The R-Tree sampling rate r determines the portion of data points in the database that will be sampled to create the R-Tree indexes. Using the entire dataset results in an R-Tree with a large branching factor and smaller MBR coverage area. A small sampling rate will create a tree with smaller branching factor but with larger MBR coverage area. This can have a trade-off in terms of indexing time and query time.

A large sampling rate can result in longer indexing time since more data points will be taken into the tree construction. But this also results in a denser tree, which can lead to more similar trajectories being in the same MBR. This can ultimately allow the query to filter out the non-candidate trajectories. A smaller sampling rate can lead to a smaller indexing time but ends up with large MBRs that contain both candidate and non-candidate trajectories, which will slow down the querying process.

Number of Spark Workers N_w

The parameter N_w determines the number of worker nodes within the distributed system. Adjusting N_w involves a trade-off between query time and index time. By increasing the number of workers, the query time can be reduced as data partitions on worker nodes are disjoint. As long as data are evenly distributed among work-

ers, adding more nodes can decrease the workload of query operations. However, increasing N_w may lead to a longer indexing time due to the communication required between all nodes during the indexing process. Consequently, adding more workers can introduce additional overhead to data partitioning.

3.1.5 Evaluation Metrics

We evaluate the indexing time and query time of PIMMLI, DITA, and the brute force approach through range and join query operations. In our range query experiment, we streamed 300 batches, each consisting of 30–90 data points for each trajectory. For the join query experiment, we streamed 20 batches of trajectory data, with each batch containing 30–90 data points for each trajectory. Each experiment was repeated 5 times, and the average index, range query, and join query times were calculated to derive the final result. The experimental results are reported by calculating the cumulative execution time in milliseconds and presented in section 3.2.

3.2 Results

3.2.1 Impact of Global Index Branching Factor N_G

We evaluate PIMMLI and DITA in terms of the indexing and query time when varying the number of global partitions N_G. We pick $N_G = \{4, 8, 16, 32, 64\}$ as the values for N_G and perform the experiment using range and join queries.

3.2.1.1 Range Query

Figure 3.2 shows that N_G has an effect on the range query time of PIMMLI. Setting $N_G = 8$ produces the best result for range query. Any value smaller or larger

than this results in higher range query time. This result shows that N_G should not be too small or too large in order to achieve the most optimal performance. $N_G = 64$ results in approximately 10.52% higher query time compared to $N_G = 8$.

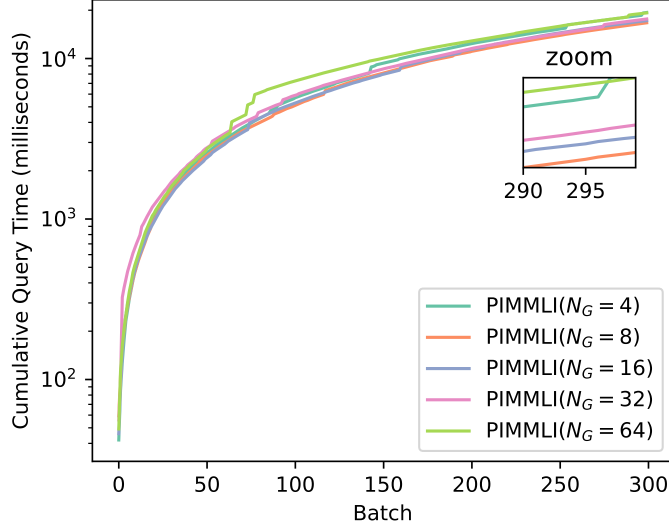

Figure 3.2: Cumulative Range Query Time of PIMMLI when Varying N_G

We also measure the query time of PIMMLI and DITA under different settings of N_G. Figure 3.3 shows that the performance of both algorithms in terms of query time are mostly similar. In all experiments that we conducted in Figure 3.3, DITA has a shorter query time compared to PIMMLI. The difference becomes bigger when we increase N_G above 8. However, the difference is negligible since DITA has only 1.18% performance improvement compared to PIMMLI on average. Both algorithms outpeformed the brute-force approach in all test cases and are approximately 14.18X faster than the brute-force method.

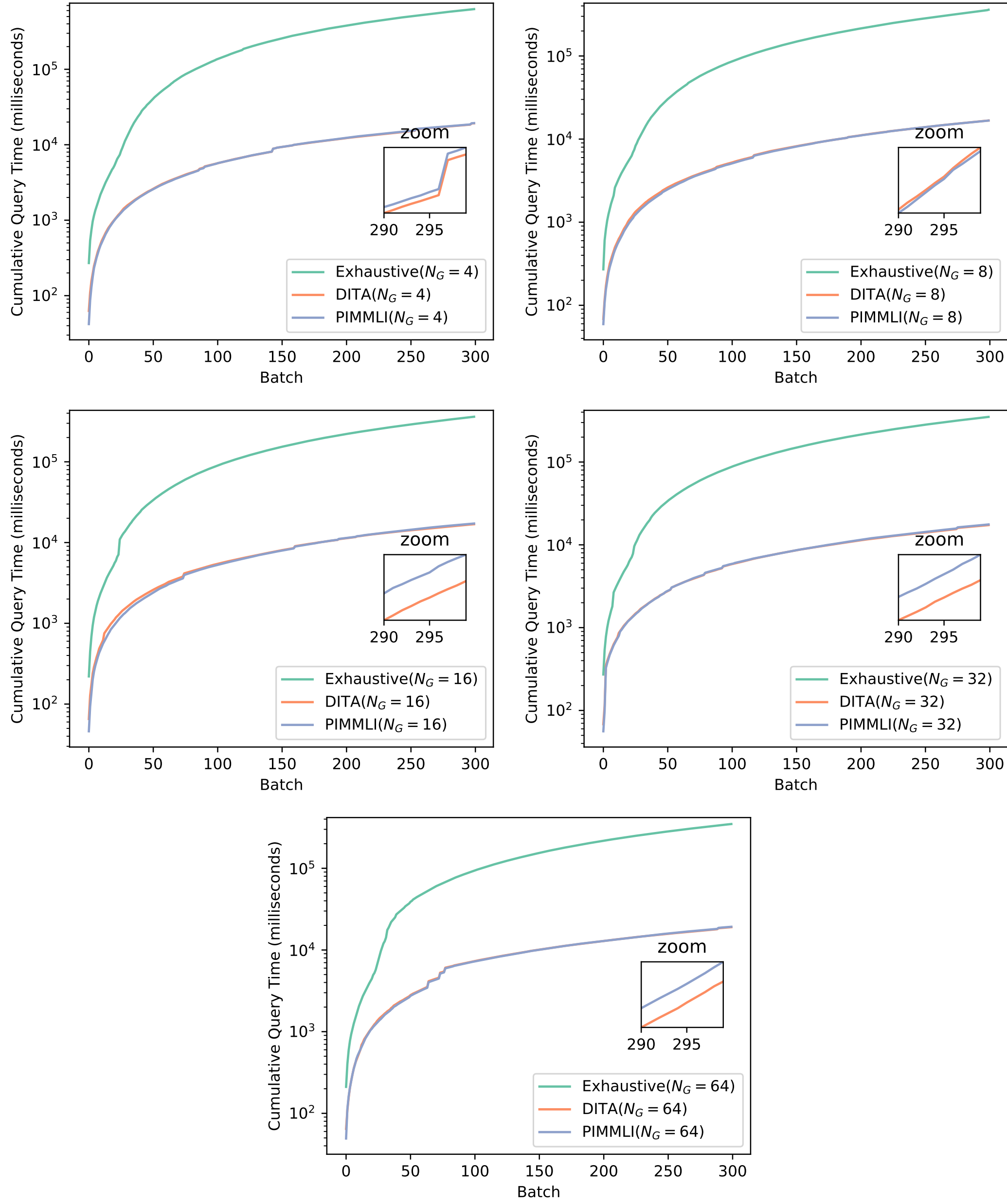

Figure 3.3: Cumulative Range Query Time of PIMMLI and DITA when Varying N_G

3.2.1.2 Join Query

Figure 3.4 shows that N_G has an effect on the join query operation. The parameter $N_G = 4$ produces the best result for join query. Increasing N_G also increases the join

query time of PIMMLI. The figure shows that larger value of N_G results in bigger join query time e.g. $N_G = 64$ has the highest query time among all testing values. $N_G = 64$ has approximately 17.61% higher query time compared to $N_G = 4$.

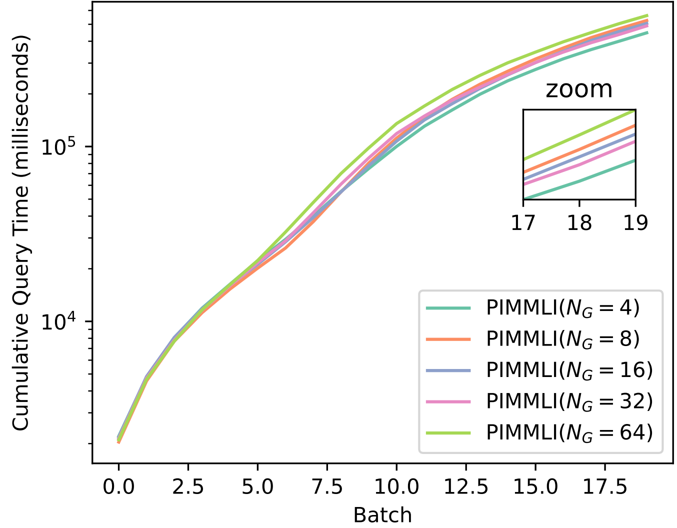

Figure 3.4: Cumulative Join Query Time of PIMMLI when Varying N_G

Figure 3.5 shows that the performance of both algorithms in terms of join query time are mostly similar. In the join query experiments that we conducted in Figure 3.5, PIMMLI performs 3.99% better than DITA with join query. However, both algorithms underperformed significantly compared to the brute-force approach. The exhaustive search is 3.63X faster than both PIMMLI and DITA.

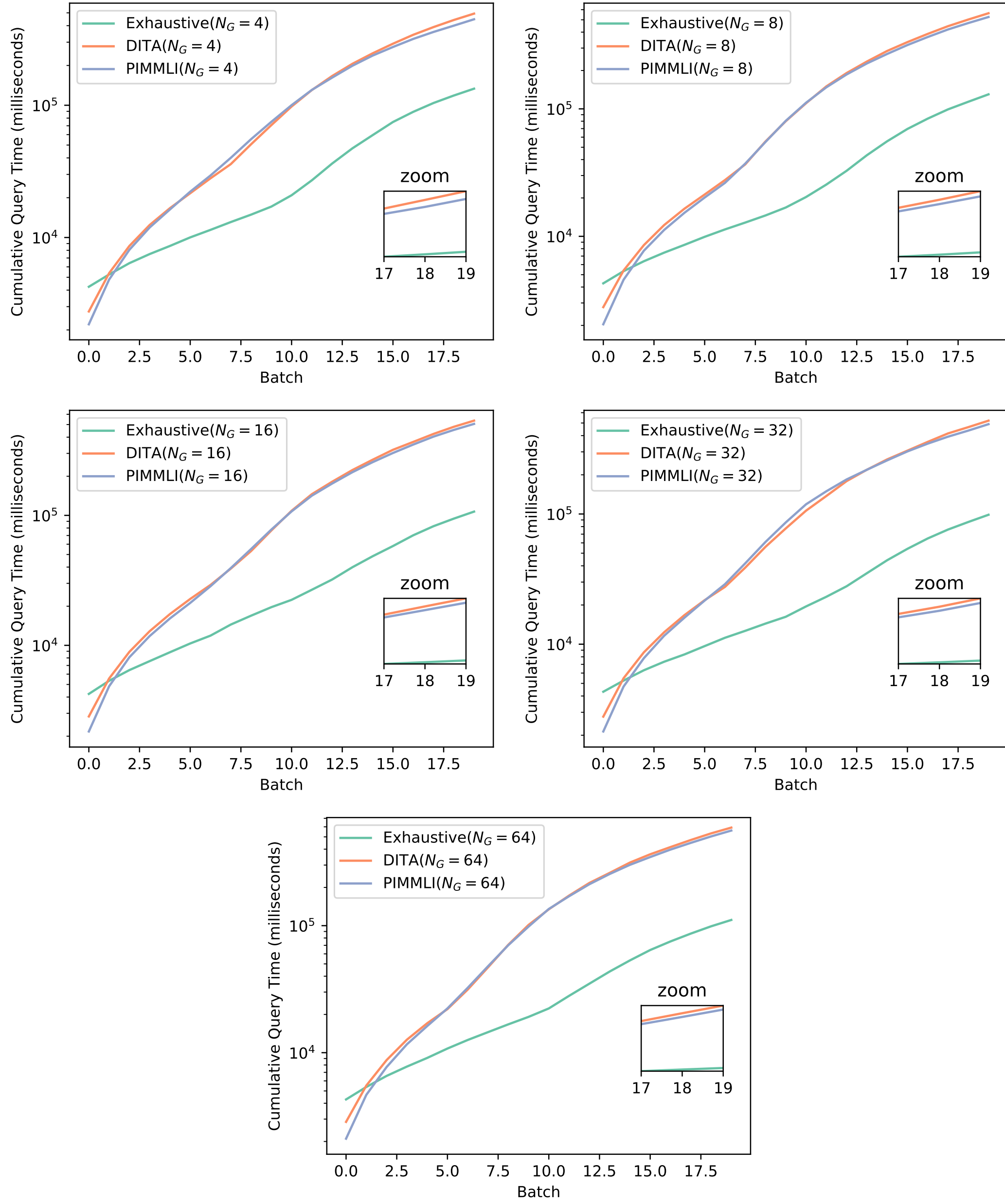

Figure 3.5: Cumulative Join Query Time of PIMMLI and DITA when Varying N_G

3.2.1.3 Index Time

Our experiment shows that $N_G = 4$ has the lowest indexing time in Figure 3.6. As we increase the value of the branching factor, the indexing time also increases. This

is expected because increasing N_G will also increase the number of global partitions which introduces additional overhead in the indexing phase. We found that $N_G = 32$ has the highest index time, approximately 25.66% higher than that of $N_G = 4$, which is about 746 seconds difference in cumulative time at batch 300. In addition, because PIMMLI uses predictive indexing, this allows the indexing process to execute less frequent than DITA. Figure 3.7 shows that PIMMLI outperformed DITA in terms of index time by approximately 27.43%.

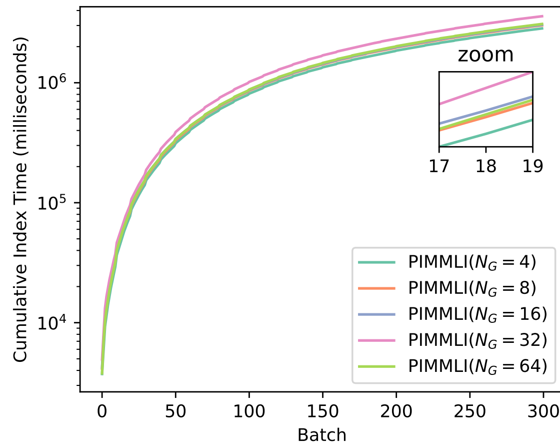

Figure 3.6: Cumulative Index Time of PIMMLI when Varying N_G

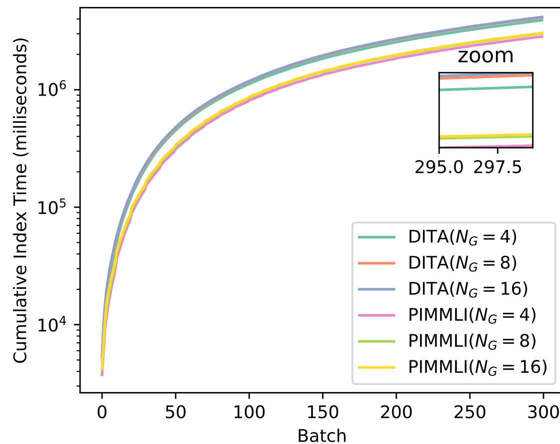

Figure 3.7: Cumulative Index Time of PIMMLI and DITA when Varying N_G

3.2.2 Impact of Local Partitions N_L

In this experiment, we study the effect of the value of local index branching factor N_L on the query operations and the index time. We use 5 different values for $N_L = \{4, 8, 16, 32, 64\}$ and discuss our findings below.

3.2.2.1 Range Query

Figure 3.8 shows PIMMLI's cumulative range query time that we obtained from varying $N_L = \{4, 8, 16, 32, 64\}$ while setting the default value for other parameters. The result that we obtained did not show significant relationship between N_L and the query time. In this experiment, $N_L = 16$ results in the highest query time, approximately 6.1% slower, while $N_L = 32$ produces the shortest query time.

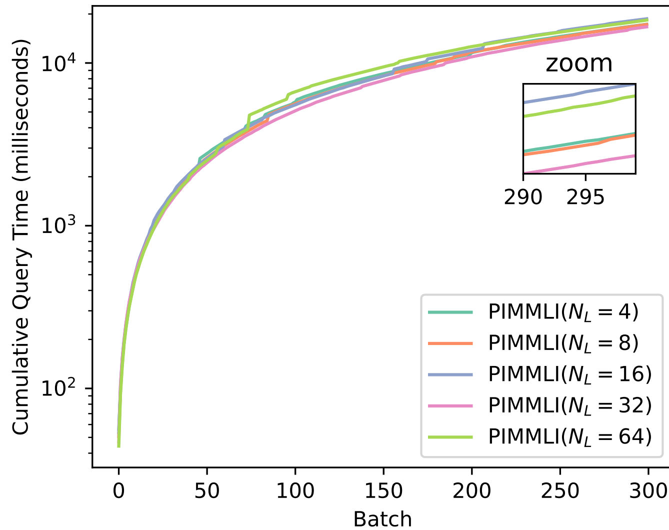

Figure 3.8: Cumulative Range Query Time of PIMMLI when Varying N_L

Figure 3.9 compares the performance of PIMMLI against DITA under different values for N_L. We found that both PIMMLI and DITA have very similar performance when performing range query operation. This result is consistent with their performances when varying N_G in 3.2.1.1. An explanation for the similarity of these per-

formances is that range query operation is not computationally expensive. Therefore, any differences in the performance of these algorithms are negligible. We compared the performance of PIMMLI against DITA at each batch and found that PIMMLI is 0.96% faster than DITA on average. In addition to that, both algorithms outperformed the exhaustive search by a factor of 13.97X.

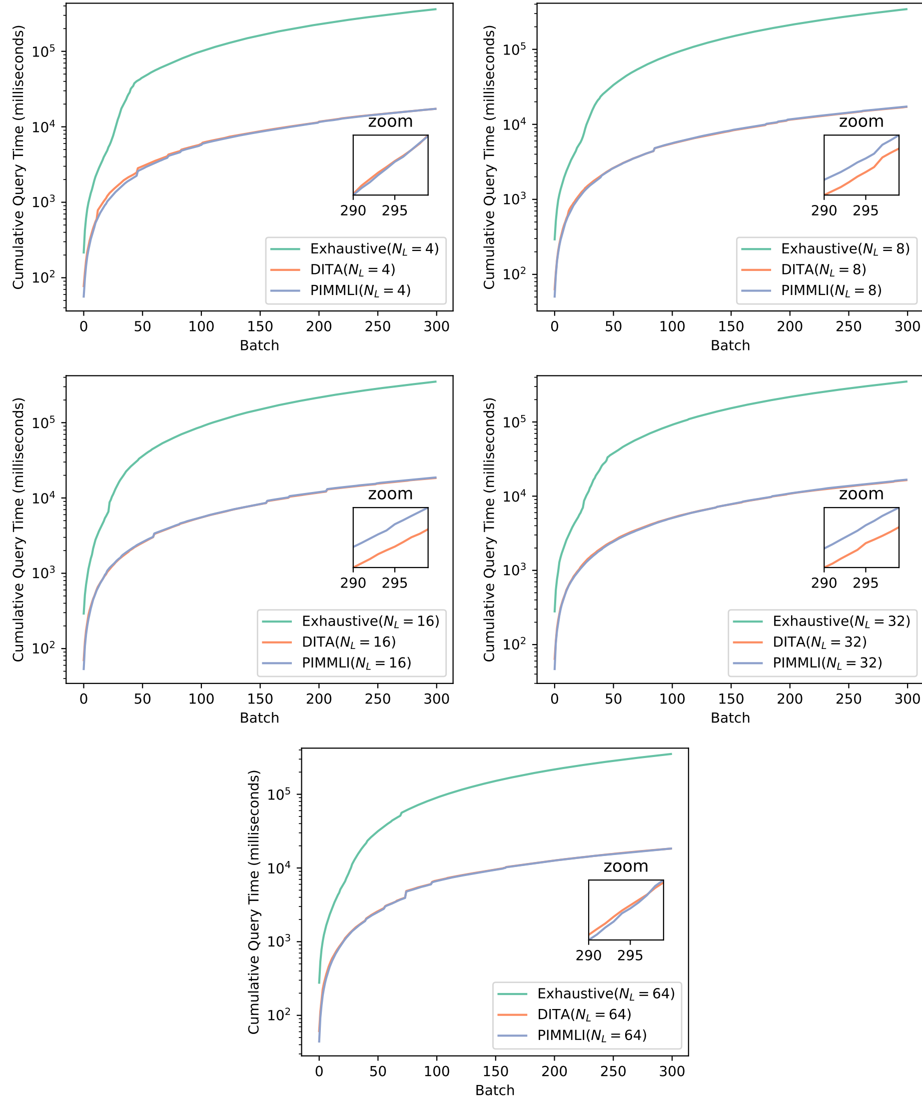

Figure 3.9: Cumulative Range Query Time of PIMMLI and DITA when Varying N_L

3.2.2.2 Join Query

In the join query experiment, we found that as the number of batches gets bigger, N_L also has a stronger effect on the query time of PIMMLI. As we double the value of N_L, the query time also increases by 2.69% on average. $N_L = 64$ has the longest query time, approximately 8.9% longer than $N_L = 4$.

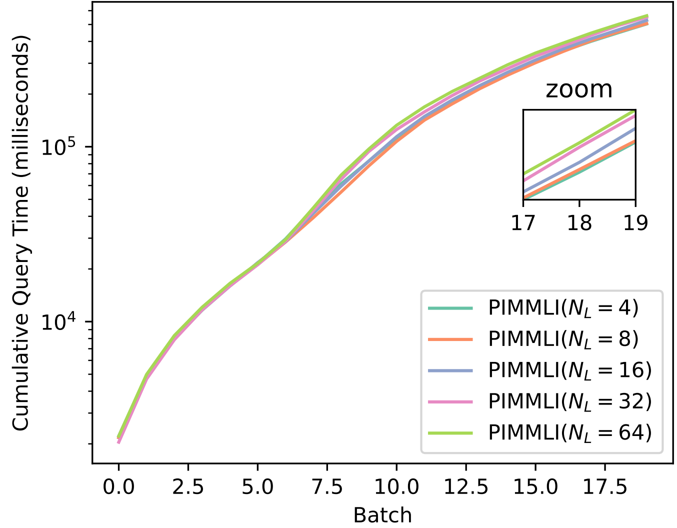

Figure 3.10: Cumulative Join Query Time of PIMMLI when Varying N_L

Our experiment also compared the performance of PIMMLI against DITA and the exhaustive search method. We found that in all cases, PIMMLI offers a slightly better performance than DITA. Our experimental result shows that PIMMLI is 3.69% faster than DITA. Although PIMMLI offers a very small improvement, it is worth noting that at batch 20, this equates to about 29 seconds difference in the cumulative query time between both algorithms. In addition to that, our result shows that the exhaustive search method outperformed both PIMMLI and DITA by an average factor of 2.41X.

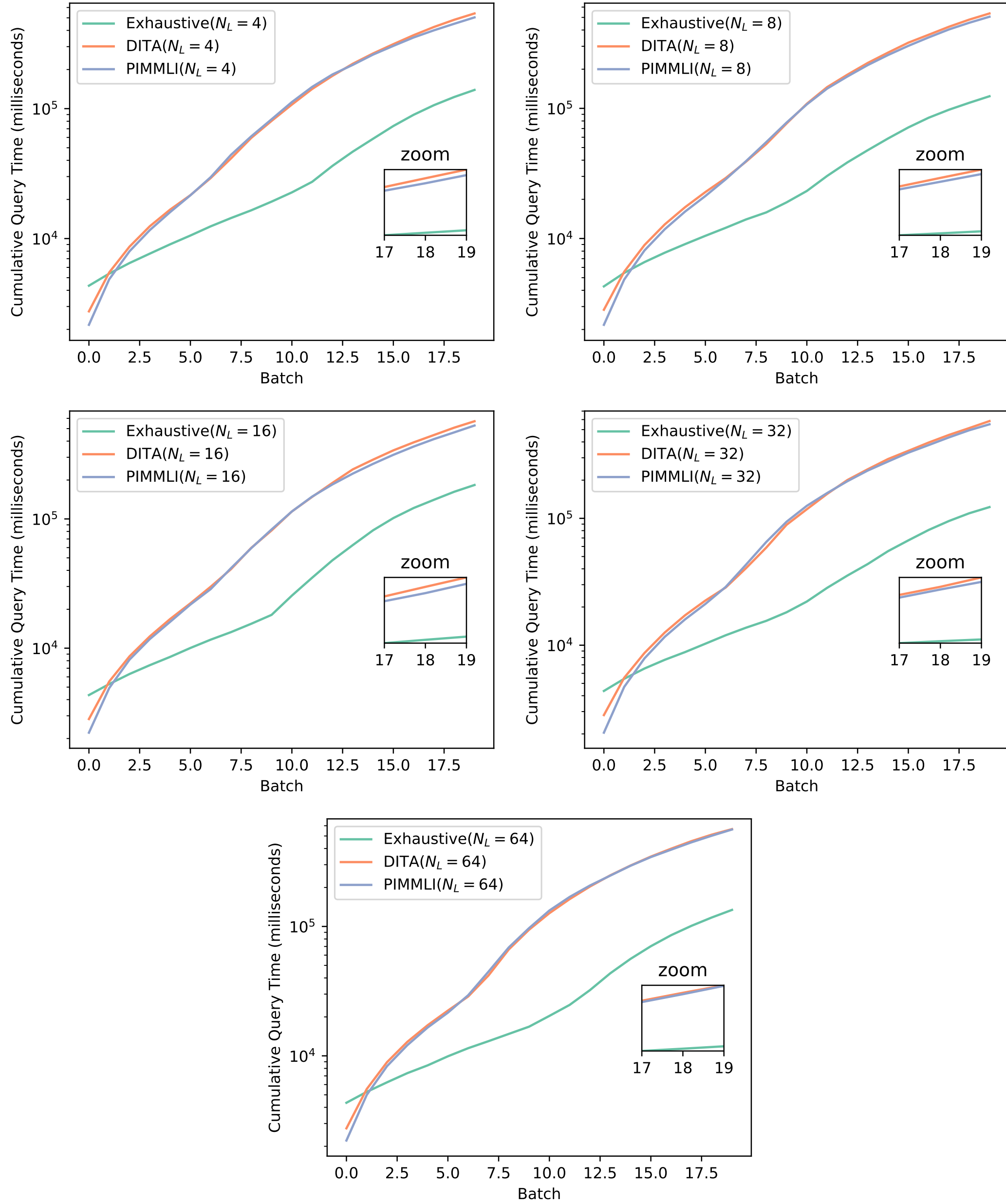

Figure 3.11: Cumulative Join Query Time of PIMMLI and DITA when Varying N_L

3.2.2.3 Index Time

Our experiment shows that $N_L = 8$ results in the lowest indexing time according to Figure 3.12. It seems that N_L does not have any strong correlation with the index

time. In addition, we found that $N_L = 4$ has the highest index time and $N_L = 8$ offers the lowest index time. $N_L = 4$ is approximately 33.19% slower than $N_L = 8$ on average, which amounts to 941 seconds at batch 300. Finally, we compared the index time of PIMMLI against DITA and presented the result in Figure 3.13. We found that PIMMLI offers a lower index time than DITA, outperformed the non-predictive version by approximately 27.74% on average.

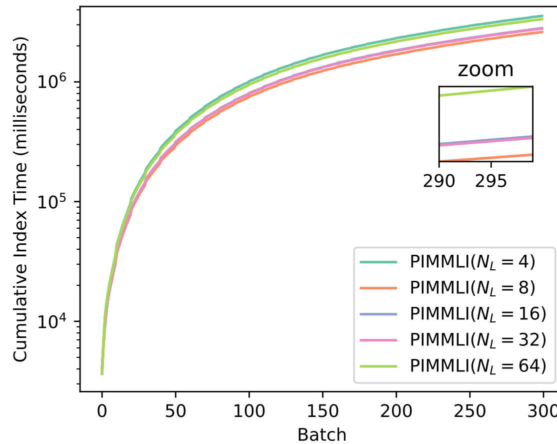

Figure 3.12: Cumulative Index Time of PIMMLI when Varying N_L

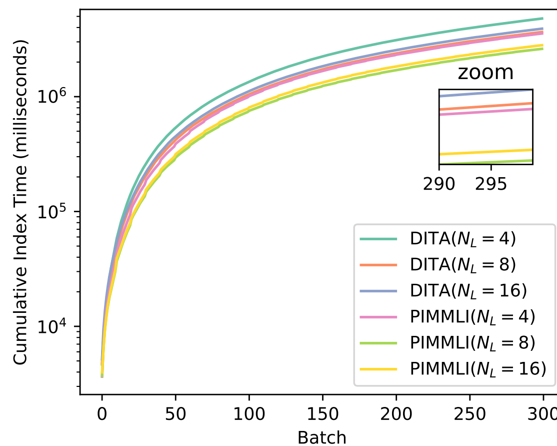

Figure 3.13: Cumulative Index Time of PIMMLI and DITA when Varying N_L

3.2.3 Impact of Sampling Rate r

We investigate the effect of the value of R-Tree sampling rate r on the algorithm. Varying r can effectively alter the structure as well as the denseness of the index tree, thus affecting the query and index time. In this experiment, we use 4 different values for $r = \{0.25, 0.5, 0.75, 1\}$ and present our findings below.

3.2.3.1 Range Query

We conducted the experiment to measure range query time of PIMMLI by varying $r = \{0.25, 0.5, 0.75, 1\}$ and presented our findings in Figure 3.14 and Figure 3.15. From the figure, we found that r does not have strong correlation with the query time. However, as we discussed previously, a low sampling rate could result in a tree with larger MBRs, which has shorter index time but with longer query time. The experimental result from Figure 3.14 supports our hypothesis. In addition to that, a denser tree seems to give better performance. All other values of r that we experimented have lower query time compared to $r = 0.25$. A sampling rate of 0.25 is roughly 13.96% slower than $r = 0.5$ on the GeoLife dataset.

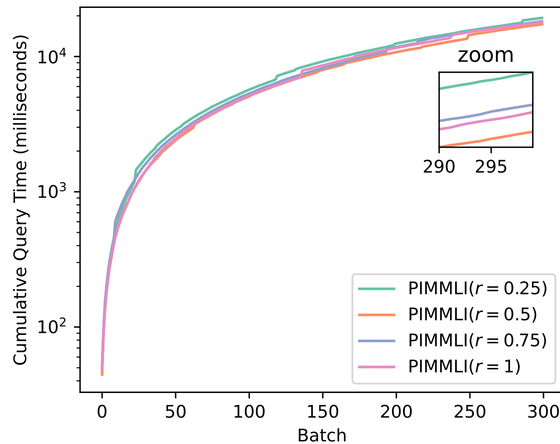

Figure 3.14: Cumulative Range Query Time of PIMMLI when Varying r

The performance comparison analysis between PIMMLI and DITA indicates that both algorithms have similar performance, with a variance of roughly 0.7% on average. In addition, both PIMMLI and DITA outperformed the exhaustive search method by a factor of 14.56X, which amounts to a difference of 354 seconds in cumulative query time at batches 300.

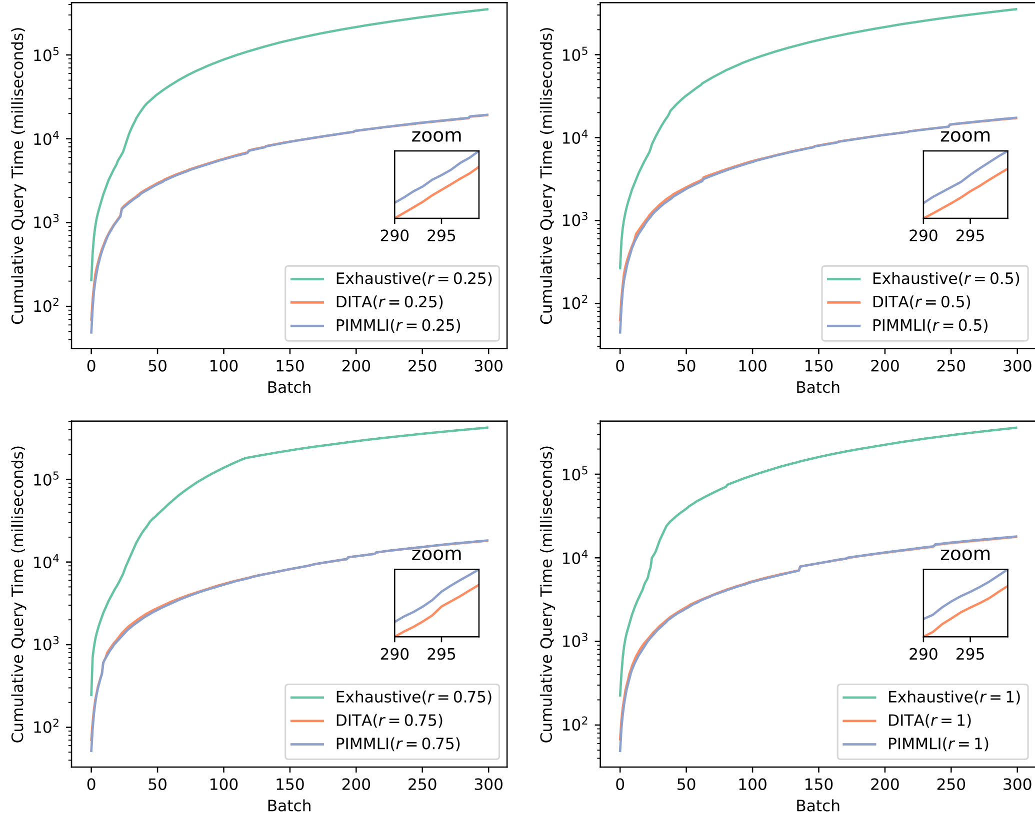

Figure 3.15: Cumulative Range Query Time of PIMMLI and DITA when Varying r

3.2.3.2 Join Query

For this experiment, we measured the run time of join query when testing different sampling rates. Our experimental result shows very similar performance of the join query regardless of the choice for r. Figure 3.16 shows that $r = 1$ seems to give the

best overall performance as opposed to $r = 0.75$. This is what we expect based on the discussion in 3.1.4. For $r = 1$, PIMMLI will sample the entire dataset to create the MBRs of the index tree, which can create a very dense tree with small MBRs. This can ultimately filter out any MBRs that contain non-candidate trajectories and reduce the execution time of the query. Our result shows that $r = 1$ is approximately 4.86% faster than $r = 0.75$.

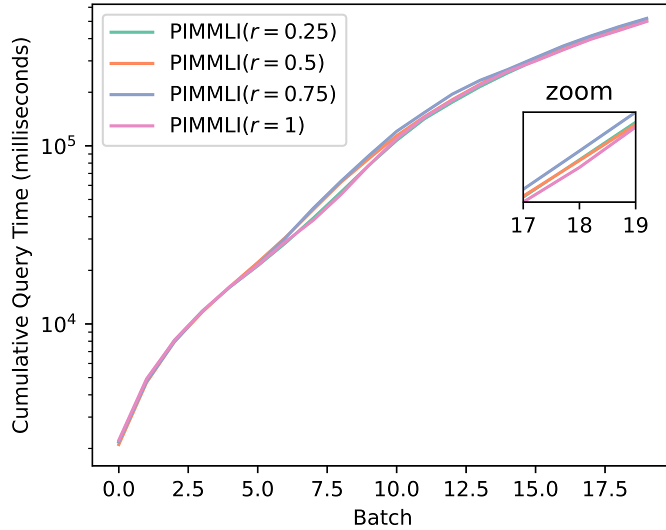

Figure 3.16: Cumulative Join Query Time of PIMMLI when Varying r

We also studied the performance of PIMMLI against DITA and the exhaustive join query. Figure 3.17 shows that PIMMLI has a similar performance compared to DITA. In addition, DITA's execution time seems to rise faster than PIMMLI as the database gets larger from our observation on the figure below. We calculated the performance of both algorithms and found that PIMMLI is 4.86% faster than DITA, which is approximately 381 seconds difference at the final batch. However, both algorithms significantly underperformed compared to the brute-force method by a factor of 2.18X.

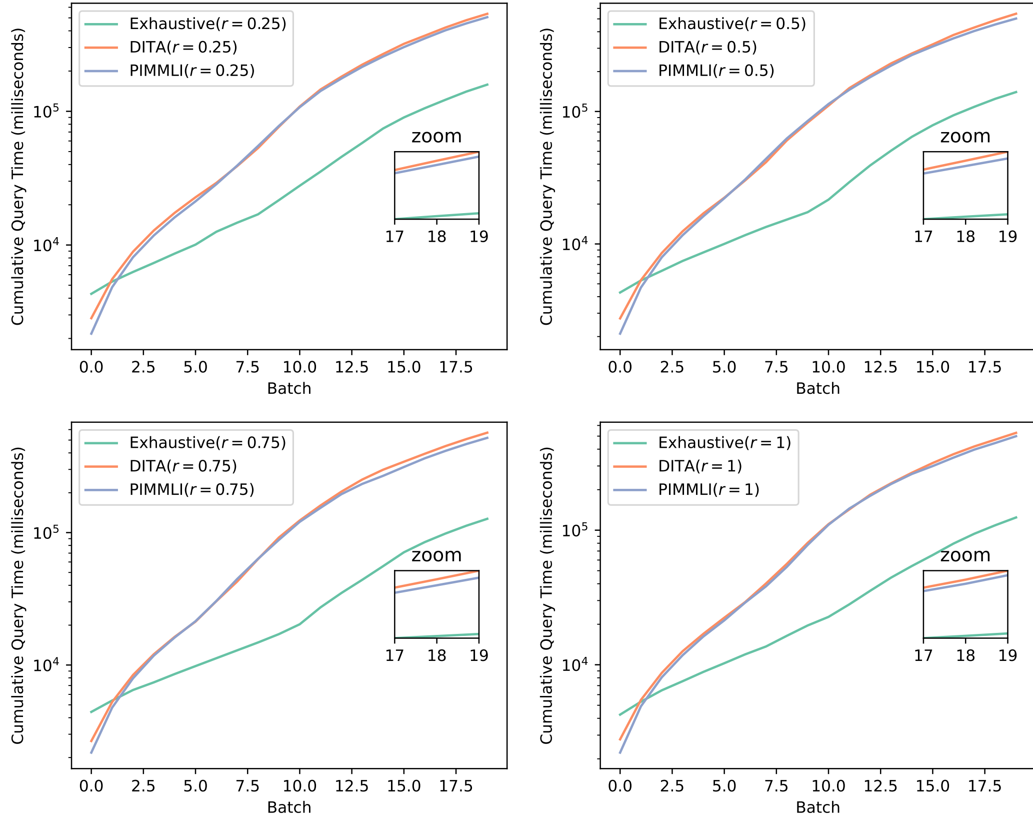

Figure 3.17: Cumulative Join Query Time of PIMMLI and DITA when Varying r

3.2.3.3 Index Time

We studied the effect of r on index time. The result that we obtained was expected from the previous discussion. As we increases r from 0.25 to 0.75, the index time also increases. This is because using more sampled data points can result in a denser tree with more MBRs. This can lead to higher index time. However, figure 3.18 shows that $r = 1$ has the lowest index time. This is due to the size of the GeoLife dataset is not sufficiently large. Therefore, using the entire dataset for indexing is more cost-effective than performing random subsampling.

Next, we compared the index time of PIMMLI against DITA. The result from Figure 3.19 shows that PIMMLI has lower index time than DITA regardless of the

choice for the sampling rate r. In addition, we computed the average execution time and found that PIMMLI is 27.99% faster than DITA in terms of index time.

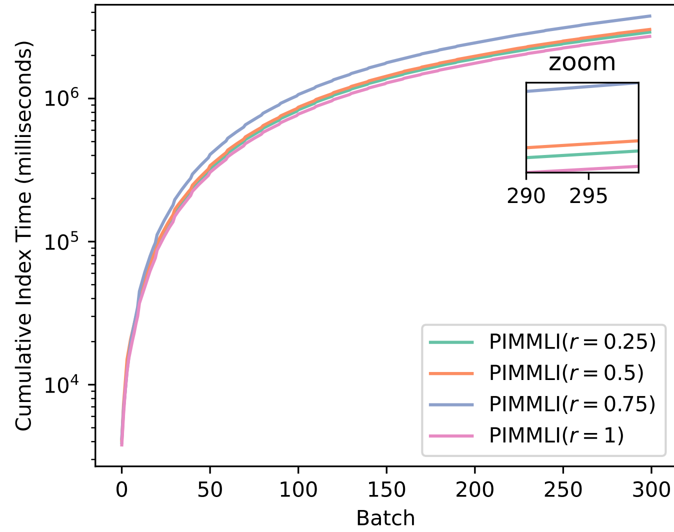

Figure 3.18: Cumulative Index Time of PIMMLI when Varying r

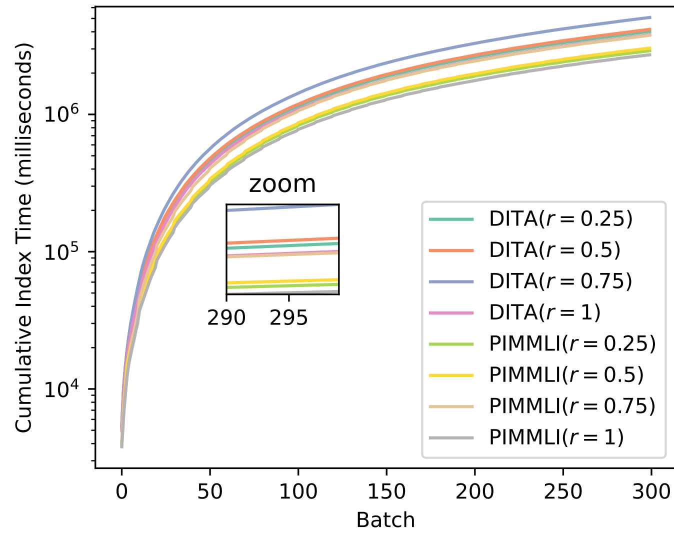

Figure 3.19: Cumulative Index Time of PIMMLI and DITA when Varying r

3.2.4 Impact of Spark Workers N_w

In this section, we discuss the performance of PIMMLI and DITA when changing the number of Spark workers in our distributed network. We measure the performance of both algorithms using $N_w = \{3, 6, 9\}$. Each worker has 16G of memory and is physically located close to one another.

3.2.4.1 Range Query

Figure 3.20 shows that the query time decreases as we increase the number of workers. Our experimental result shows that doubling the number of workers reduces the query time of PIMMLI by approximately 4.59%. This result is in line with what we discussed previously in section 3.1.4. Since trajectory search operations are independent of each other, increasing the number of workers can, in turn, decrease the query time.

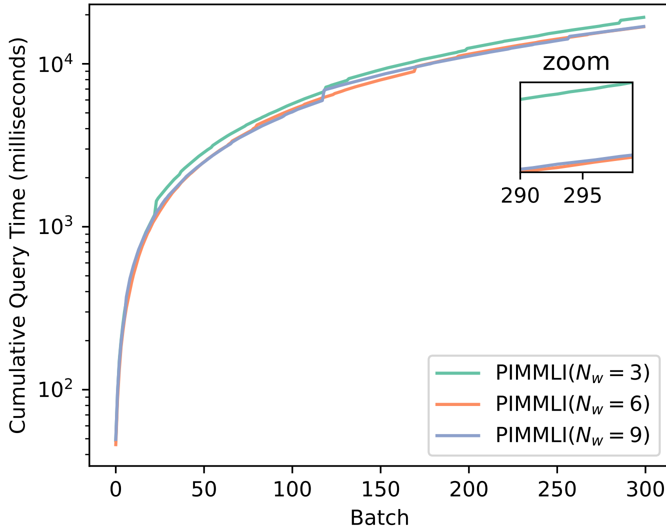

Figure 3.20: Cumulative Range Query Time of PIMMLI when Varying N_w

Figure 3.21 presents a comparison of the range query time of PIMMLI against DITA. Both algorithms exhibit very similar performance. PIMMLI is only 0.86%

slower than DITA. Furthermore, both algorithms outperformed the brute-force query method by a factor of 16.79X.

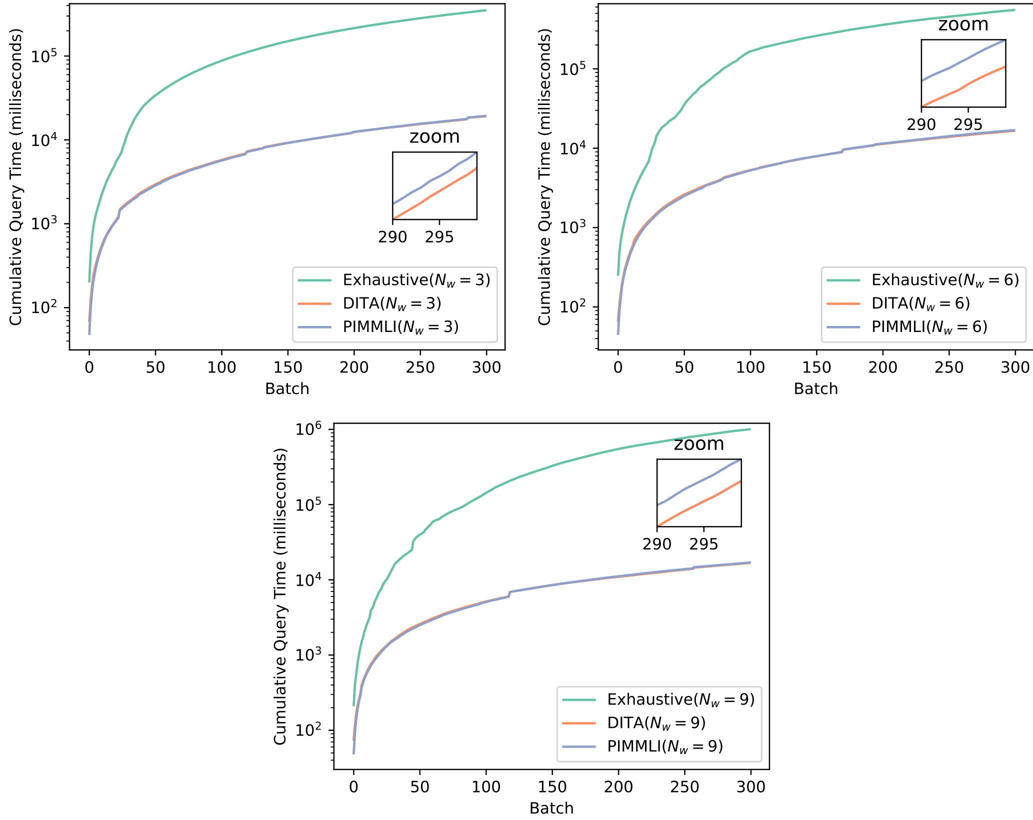

Figure 3.21: Cumulative Range Query Time of PIMMLI and DITA when Varying N_w

3.2.4.2 Join Query

Next, we studied the effect of N_w on the execution time of join query. The result that we obtained is also in accordance with the result of range query. As we increase the number of worker nodes, the join query time decrease by approximately 12.8% on average. In addition to that, the effect seems to diminish when N_w is above 6. We then compared PIMMLI against DITA and the exhaustive method. Both PIMMLI and DITA have very similar join query performance. However, they all underperformed

the exhaustive search by a factor of 1.91X.

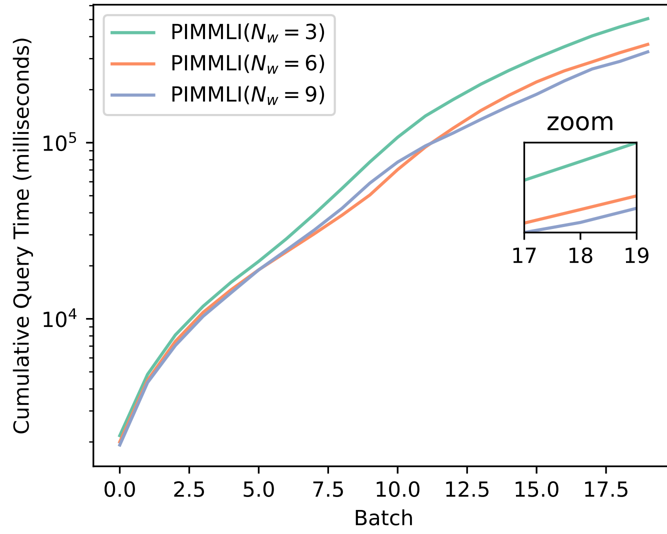

Figure 3.22: Cumulative Join Query Time of PIMMLI when Varying N_w

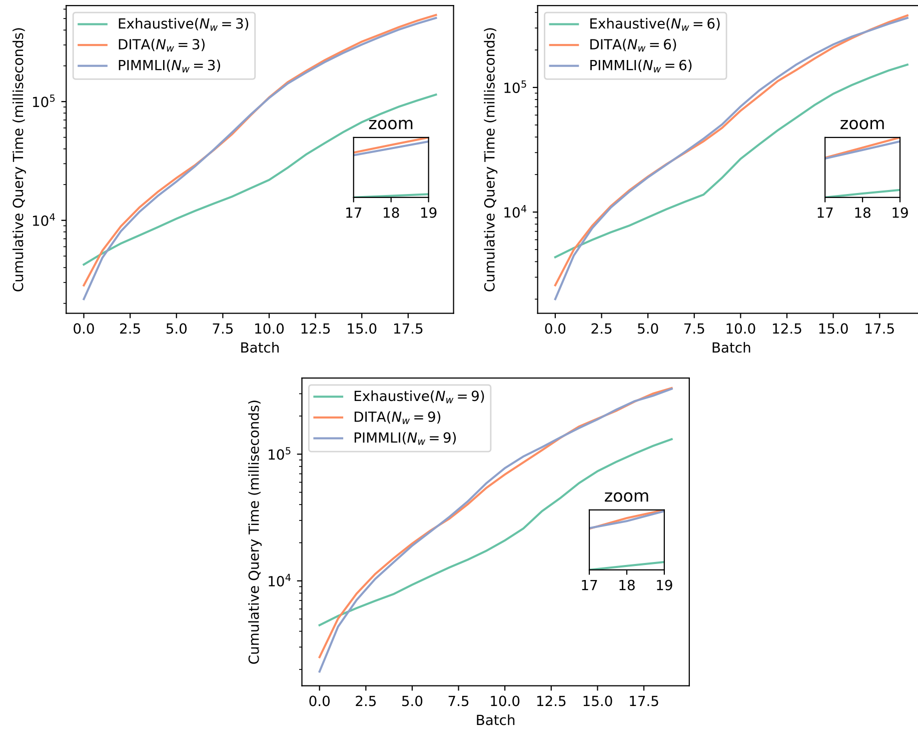

Figure 3.23: Cumulative Join Query Time of PIMMLI and DITA when Varying N_w

3.2.4.3 Index Time

Finally, the result from Figure 3.24 shows that the index time increases as we increase the number of Spark workers. The effect of N_w on the index time starts becoming more evident when the amount of indexed batches gets larger. Our result shows that increasing N_w will increase the indexing time of PIMMLI by roughly 8.94%. This result is in accordance with what we expect from the previous discussion. Since the index phase requires communication between every node in the network, increasing N_w introduces additional overhead in the distributed system, which can, in turn, increase the index time of the trajectory data.

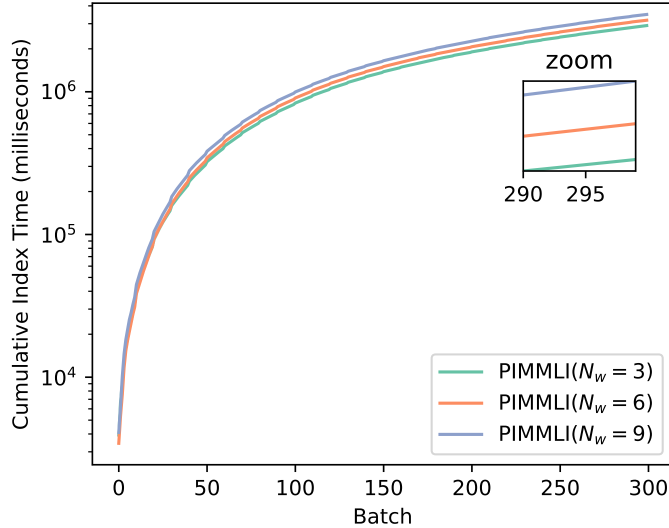

Figure 3.24: Cumulative Index Time of PIMMLI when Varying N_w

Lastly, figure 3.25 shows that N_w has a proportional effect on the index time of both algorithms. However, PIMMLI is approximately 29.22% faster than DITA in terms of index time. This is because PIMMLI uses predictive modeling during the index phase, which allows the size of the MBRs to capture the unseen future points a few steps ahead. This can ultimately allow PIMMLI to perform the data indexing less frequent than DITA.

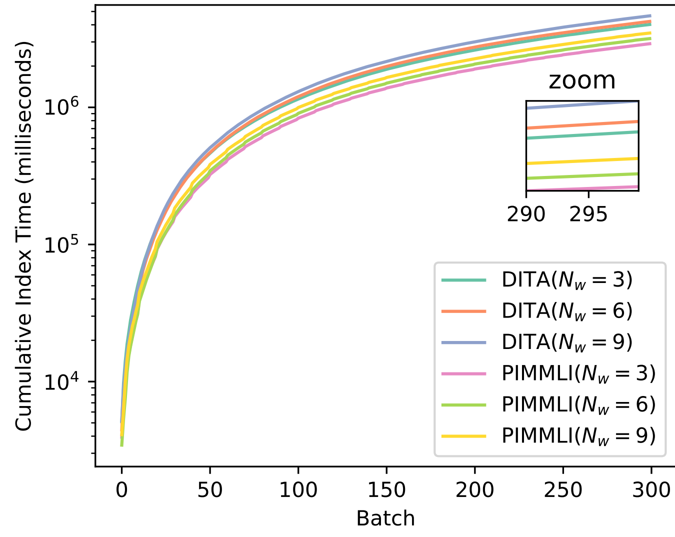

Figure 3.25: Cumulative Index Time of PIMMLI and DITA when Varying N_w

4 Conclusions & Future Work

In this chapter, we summarize the experimental results from Section 3.2 and outline future work to conduct more in-depth analyses and improvements on PIMMLI.

4.1 Conclusions

- We introduced PIMMLI, the first predictive trajectory analysis algorithm that predicts future points for data indexing. PIMMLI is highly scalable by using Apache Spark — a framework for large-scale data processing in distributed streaming environments. PIMMLI implements an R-Tree data structure to index the spatial trajectories from the data stream.
- In addition, PIMMLI builds multiple predictive models for each trajectory to predict their future points. This allows it to expand the minimum bounding rectangles (MBRs) that cover the data points ahead of time, which can reduce the index time significantly. However, this can also creates a trade-off between time and space complexity. A larger MBR can encompass a greater number of data points within the trajectories, enabling less frequent data indexing. However, this may lead to an imbalance indexing, where one partition contains more trajectories than another. This can ultimately affect the performance of a query.
- Both PIMMLI and DITA outperformed the exhaustive search method in range

queries by a factor of 14.88X. However, we noticed that both algorithms underperformed significantly compared to the exhaustive method in join queries. This could be because we conducted the experiment with only 20 batches due to the high cost of join query. Therefore, the data that we collected was not enough to reflect the correct performance of PIMMLI and DITA against the brute-force method.

Range and Join Query

- The choice of N_G depends on the selected dataset and has some effects on the range query time. Picking a small value for N_G seems to result in better performance. We observed that an optimal choice for N_G can reduce the query time by approximately 9.52%. However, N_G does not have a strong improvement on join query time.
- We observed an average improvement of 5.74% when selecting an optimal value for N_L. We found that picking a large value for N_L can result in the best performance for range query. A value of $N_L = 32$ has the best performance for range queries at every batch iteration during our experiment. However, that effect diminishes as N_L gets too large. In the join query experiment, we found that N_L does not have strong effect on the query time.
- We conducted an experiment to study the impact of the random sampling rate r and found that an optimal value for r can reduce the range query time by roughly 12.25% using the GeoLife dataset. We observed that $r = 0.5$ achieve the best overall performance. A low value for r will result in an R-Tree with low branching factor. This has a poor performance on the query time because both

of the candidate and non-candidate trajectories can fall into the same MBR. As for the join query, $r = 1$ gives the best performance, approximately 4.86% faster than the non-optimal choice for $r = 0.75$.

- Last but least, because queries can be executed on the partitions independently, we expect that increasing the number of worker nodes can, in turn, decrease their query time. Our experiments confirmed this. As we increase the number of nodes, the range and join query time decreases by 4.59% and 12.8% on average, respectively. However, we also noticed that the effect is smaller when N_w is larger than 6.

Trajectory Indexing

- Firstly, we observed that a small value for N_G seems to give the lowest indexing time. This is because increasing N_G also increases the number of global partitions, which will introduce additional overhead. As a rule of thumb, N_G should be set to a small value for an optimal index time.
- Secondly, we found that picking $N_L = 8$ gives the most optimal index time. A local branching factor that is either too small or too large will result in very high index time, approximately 33.19% higher than when $N_L = 8$.
- Thirdly, the random sampling rate r has a strong effect on the index time. As we increase the sampling rate, the index time also increases, respectively. However, we noticed that $r = 1$ gave the lowest index time. This is because our experimental dataset GeoLife is not large enough. Therefore, using the entire dataset to build the MBRs gave the best performance.

- Fourthly, we measured the index time as we vary the number of worker nodes. We found that the value of N_w is positively correlated with the index time of PIMMLI. Setting $N_w = 3$ gives the most optimal performance and the index time increases roughly 8.94% every time we add 3 more nodes into the network.

- Lastly, PIMMLI has a significant lower index time compared to DITA. This is because PIMMLI employs a predictive indexing technique that allows expanding the coverage area of the MBRs ahead of time. This allows the algorithm to rebuild the R-Tree less frequent than DITA. In all experiments, we found that PIMMLI is roughly 28.10% faster than DITA in terms of indexing time on average.

4.2 Future Work

- We introduced PIMMLI, the first trajectory analysis algorithm that uses predictive modeling to reduce the index time. We implemented a simple trajectory prediction model and reduced the index time by roughly 28.10%. It maybe interesting to analyze the performance of more complex predictive models to study the trade-off between the accuracy and the index time.

- By predicting the future points of the trajectories, PIMMLI can expand the area of the MBRs that are covering these trajectories. Even though this helps reduce the index time, it is important to note that this can affect the space complexity of the algorithm. It may also be interesting to study the trade-off between time and space complexity.

- Next, it may be interesting to compare PIMMLI against other state-of-the-art trajectory analysis algorithms to further understand the benefits and drawbacks

of predictive modeling in trajectory indexing.

- Next, we mentioned the use of different similarity measures in trajectory analysis algorithms in subsection 1.1.3. PIMMLI uses DTW to measure the similarity between two trajectories. It may be interesting to see how the algorithm performs when using other existing similarity metrics.

- Finally, the development of a distributed trajectory analysis algorithm is a very important task. One advantage of distributed systems over centralized systems is the vertical and horizontal scalability. However, distributed systems often come with network overhead compared to centralized systems. We saw this affecting the performance of PIMMLI in 3.2.4 in which increasing the number of nodes can reduce the query time but at the same time, increase the index time. It may be interesting to conduct further experiments to study PIMMLI in both environments and address the benefits as well as the drawbacks of the two computational systems.

References

[1] A. Wägli, “Trajectory determination and analysis in sports by satellite and inertial navigation,” EPFL, Tech. Rep., 2009 (cit. on p. 1).

[2] N. Marković, P. Sekuła, Z. Vander Laan, G. Andrienko, and N. Andrienko, “Applications of trajectory data from the perspective of a road transportation agency: Literature review and maryland case study,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 5, pp. 1858–1869, 2018 (cit. on p. 1).

[3] M. Moreira-Soares, E. Mossmann, R. D. Travasso, and J. R. Bordin, “Trajpy: Empowering feature engineering for trajectory analysis across domains,” *Bioinformatics Advances*, vol. 4, no. 1, vbae026, 2024 (cit. on p. 1).

[4] G. Weng, J. Kim, and K. J. Won, “Vetra: A tool for trajectory inference based on rna velocity,” *Bioinformatics*, vol. 37, no. 20, pp. 3509–3513, 2021 (cit. on p. 1).

[5] Y. Zhang, J. Liu, X. Qian, A. Qiu, and F. Zhang, “An automatic road network construction method using massive gps trajectory data,” *ISPRS International Journal of Geo-Information*, vol. 6, no. 12, p. 400, 2017 (cit. on p. 1).

[6] Z. Feng and Y. Zhu, "A survey on trajectory data mining: Techniques and applications," *IEEE Access*, vol. 4, pp. 2056–2067, 2016. DOI: 10.1109/ACCESS.2016.2553681 (cit. on p. 1).

[7] H. Jeung, H. Lu, S. Sathe, and M. L. Yiu, "Managing evolving uncertainty in trajectory databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 7, pp. 1692–1705, 2014. DOI: 10.1109/TKDE.2013.141 (cit. on p. 2).

[8] J. Bao, Y. Zheng, and M. F. Mokbel, "Location-based and preference-aware recommendation using sparse geo-social networking data," in *Proceedings of the 20th international conference on advances in geographic information systems*, 2012, pp. 199–208 (cit. on p. 2).

[9] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005 (cit. on p. 3).

[10] J. Mao, Q. Song, C. Jin, Z. Zhang, and A. Zhou, "Online clustering of streaming trajectories," *Frontiers of Computer Science*, vol. 12, pp. 245–263, 2018 (cit. on p. 3).

[11] K. Patroumpas, "Multi-scale window specification over streaming trajectories," *Journal of Spatial Information Science*, no. 7, pp. 45–75, 2013 (cit. on p. 3).

[12] Y. Tao, A. Both, R. I. Silveira, *et al.*, "A comparative analysis of trajectory similarity measures," *GIScience & Remote Sensing*, vol. 58, no. 5, pp. 643–669, 2021 (cit. on p. 3).

[13] P. e. Senin, "Dynamic time warping algorithm review," *Information and Computer Science Department University of Hawaii at Manoa Honolulu, USA*, vol. 855, no. 1-23, p. 40, 2008 (cit. on pp. 3, 11).

[14] H. Sakoe, “Two-level dp-matching–a dynamic programming-based pattern matching algorithm for connected word recognition,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 27, no. 6, pp. 588–595, 1979 (cit. on p. 3).

[15] C. S. Myers and L. R. Rabiner, “A comparative study of several dynamic time-warping algorithms for connected-word recognition,” *Bell System Technical Journal*, vol. 60, no. 7, pp. 1389–1409, 1981 (cit. on p. 3).

[16] Y. Zheng, “Trajectory data mining: An overview,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 3, pp. 1–41, 2015 (cit. on p. 4).

[17] Z. Feng and Y. Zhu, “A survey on trajectory data mining: Techniques and applications,” *IEEE Access*, vol. 4, pp. 2056–2067, 2016 (cit. on p. 4).

[18] J. Qin, G. Mei, and L. Xiao, “Building the traffic flow network with taxi gps trajectories and its application to identify urban congestion areas for traffic planning,” *Sustainability*, vol. 13, no. 1, p. 266, 2020 (cit. on p. 4).

[19] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, “Closest pair queries in spatial databases,” *ACM SIGMOD Record*, vol. 29, no. 2, pp. 189–200, 2000 (cit. on p. 5).

[20] G. Zachár, “Visualization of large-scale trajectory datasets,” in *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, 2023, pp. 152–157 (cit. on p. 5).

[21] A. Guttman, “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 (cit. on p. 6).

[22] S. Srividhya and S. Lavanya, "Comparative analysis of r-tree and r-tree in spatial database," in *2014 International Conference on Intelligent Computing Applications*, IEEE, 2014, pp. 449–453 (cit. on p. 6).

[23] M. M. Sardadi, M. S. bin Mohd Rahim, Z. Jupri, and D. bin Daman, "Choosing r-tree or quadtree spatial dataindexing in one oracle spatial database system to make faster showing geographical map in mobile geographical information system technology," *International Journal of Computer and Information Engineering*, vol. 2, no. 10, pp. 3345–3353, 2008 (cit. on p. 6).

[24] G. Proietti and C. Faloutsos, "Analysis of range queries and self-spatial join queries on real region datasets stored using an r-tree," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 5, pp. 751–762, 2000 (cit. on p. 6).

[25] M. A. Abd Elwahab, K. M. Mahar, H. Abdelkader, and H. A. Khater, "Combining r-tree and b-tree to enhance spatial queries processing," in *2013 23rd International Conference on Computer Theory and Applications (ICCTA)*, IEEE, 2013, pp. 184–190 (cit. on p. 6).

[26] Y. Zheng and X. Zhou, *Computing with spatial trajectories*. Springer Science & Business Media, 2011 (cit. on p. 7).

[27] Z. Fang, L. Chen, Y. Gao, L. Pan, and C. S. Jensen, "Dragoon: A hybrid and efficient big trajectory management system for offline and online analytics," *The VLDB Journal*, vol. 30, pp. 287–310, 2021 (cit. on pp. 8, 10, 13).

[28] C. Cai and D. Lin, "Find another me across the world-large-scale semantic trajectory analysis using spark," *IEEE Transactions on Knowledge and Data Engineering*, 2022 (cit. on p. 8).

[29] W. Xiong, X. Wang, and H. Li, "Efficient large-scale gps trajectory compression on spark: A pipeline-based approach," *Electronics*, vol. 12, no. 17, p. 3569, 2023 (cit. on p. 8).

[30] M. S. Mahmud, J. Z. Huang, S. Salloum, T. Z. Emara, and K. Sadatdiynov, "A survey of data partitioning and sampling methods to support big data analysis," *Big Data Mining and Analytics*, vol. 3, no. 2, pp. 85–101, 2020 (cit. on p. 8).

[31] B. Shangguan, P. Yue, Z. Wu, and L. Jiang, "Big spatial data processing with apache spark," in *2017 6th International Conference on Agro-Geoinformatics*, IEEE, 2017, pp. 1–4 (cit. on p. 8).

[32] Z. Huang, Y. Chen, L. Wan, and X. Peng, "Geospark sql: An effective framework enabling spatial queries on spark," *ISPRS International Journal of Geo-Information*, vol. 6, no. 9, p. 285, 2017 (cit. on p. 8).

[33] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on apache spark," *International Journal of Data Science and Analytics*, vol. 1, pp. 145–164, 2016 (cit. on p. 9).

[34] A. Elomari, A. Maizate, and L. Hassouni, "Data storage in big data context: A survey," in *2016 Third International Conference on Systems of Collaboration (SysCo)*, IEEE, 2016, pp. 1–4 (cit. on p. 9).

[35] S. Wang, Z. Bao, J. S. Culpepper, Z. Xie, Q. Liu, and X. Qin, "Torch: A search engine for trajectory data," in *The 41st international ACM SIGIR conference on research & development in information retrieval*, 2018, pp. 535–544 (cit. on pp. 9, 10).

[36] Y. Yang, J. Cai, H. Yang, J. Zhang, and X. Zhao, "Tad: A trajectory clustering algorithm based on spatial-temporal density analysis," *Expert Systems with Applications*, vol. 139, p. 112 846, 2020 (cit. on pp. 9, 10).

[37] D. Birant and A. Kut, "St-dbscan: An algorithm for clustering spatial–temporal data," *Data & knowledge engineering*, vol. 60, no. 1, pp. 208–221, 2007 (cit. on pp. 9, 10).

[38] C. A. Ferrero, L. O. Alvares, and V. Bogorny, "Multiple aspect trajectory data analysis: Research challenges and opportunities.," *GeoInfo*, pp. 56–67, 2016 (cit. on p. 10).

[39] G. Yuan, P. Sun, J. Zhao, D. Li, and C. Wang, "A review of moving object trajectory clustering algorithms," *Artificial Intelligence Review*, vol. 47, pp. 123–144, 2017 (cit. on p. 10).

[40] A. Belhadi, Y. Djenouri, J. C.-W. Lin, and A. Cano, "Trajectory outlier detection: Algorithms, taxonomies, evaluation, and open challenges," *ACM Transactions on Management Information Systems (TMIS)*, vol. 11, no. 3, pp. 1–29, 2020 (cit. on p. 10).

[41] Z. Shang, G. Li, and Z. Bao, "Dita: Distributed in-memory trajectory analytics," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, Houston, TX, USA: Association for Computing Machinery, 2018, pp. 725–740, ISBN: 9781450347037. DOI: 10.1145/3183713.3183743. [Online]. Available: https://doi.org/10.1145/3183713.3183743 (cit. on pp. 10, 12).

[42] P. Cudre-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in *2010 IEEE 26th International*

Conference on Data Engineering (ICDE 2010), IEEE, 2010, pp. 109–120 (cit. on p. 10).

[43] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, and S. Sadiq, "Sharkdb: An in-memory column-oriented trajectory storage," in *Proceedings of the 23rd ACM international conference on conference on information and knowledge management*, 2014, pp. 1409–1418 (cit. on p. 10).

[44] S. T. Leutenegger, M. A. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for r-tree packing," in *Proceedings 13th international conference on data engineering*, IEEE, 1997, pp. 497–506 (cit. on p. 10).

[45] X. Fan, B. Li, and S. Sisson, "The binary space partitioning-tree process," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2018, pp. 1859–1867 (cit. on p. 10).

[46] H. Vo, A. Aji, and F. Wang, "Sato: A spatial data partitioning framework for scalable query processing," in *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*, 2014, pp. 545–548 (cit. on p. 10).

[47] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander, "Pist: An efficient and practical indexing technique for historical spatio-temporal point data," *GeoInformatica*, vol. 12, pp. 143–168, 2008 (cit. on p. 10).

[48] C. Düntgen, T. Behr, and R. H. Güting, "Berlinmod: A benchmark for moving object databases," *The VLDB Journal*, vol. 18, pp. 1335–1368, 2009 (cit. on p. 10).

[49] K. Toohey and M. Duckham, "Trajectory similarity measures," *Sigspatial Special*, vol. 7, no. 1, pp. 43–50, 2015 (cit. on p. 11).

[50] B.-K. Yi, H. V. Jagadish, and C. Faloutsos, "Efficient retrieval of similar time sequences under time warping," in *Proceedings 14th International Conference on Data Engineering*, IEEE, 1998, pp. 201–208 (cit. on p. 11).

[51] L. Chen and R. Ng, "On the marriage of lp-norms and edit distance," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 792–803 (cit. on p. 11).

[52] A. A. Taha and A. Hanbury, "An efficient algorithm for calculating the exact hausdorff distance," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 11, pp. 2153–2163, 2015 (cit. on p. 11).

[53] M. Vlachos, G. Kollios, and D. Gunopulos, "Discovering similar multidimensional trajectories," in *Proceedings 18th international conference on data engineering*, IEEE, 2002, pp. 673–684 (cit. on p. 11).

[54] S. Ranu, P. Deepak, A. D. Telang, P. Deshpande, and S. Raghavan, "Indexing and matching trajectories under inconsistent sampling rates," in *2015 IEEE 31st International conference on data engineering*, IEEE, 2015, pp. 999–1010 (cit. on p. 11).

[55] S. Wang, Z. Bao, J. S. Culpepper, and G. Cong, "A survey on trajectory data management, analytics, and learning," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–36, 2021 (cit. on p. 11).

[56] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 1071–1085 (cit. on pp. 11, 12).

[57] D. Xie, F. Li, and J. M. Phillips, "Distributed trajectory similarity search," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1478–1489, 2017 (cit. on p. 12).

[58] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao, "Ultraman: A unified platform for big trajectory data management and analytics," *Proceedings of the VLDB Endowment*, vol. 11, no. 7, pp. 787–799, 2018 (cit. on pp. 12, 13).

[59] A. Samad, "Pimmli: Predictive in-memory multi-level indexing for distributed trajectory streams," Ph.D. dissertation, University of Minnesota, 2021 (cit. on p. 13).

[60] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma, "Mining interesting locations and travel sequences from gps trajectories," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 791–800 (cit. on p. 32).

[61] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma, "Understanding mobility based on gps data," in *Proceedings of the 10th international conference on Ubiquitous computing*, 2008, pp. 312–321 (cit. on p. 32).

[62] Y. Zheng, X. Xie, W.-Y. Ma, *et al.*, "Geolife: A collaborative social networking service among user, location and trajectory.," *IEEE Data Eng. Bull.*, vol. 33, no. 2, pp. 32–39, 2010 (cit. on p. 32).