PIMMLI: Predictive In-Memory Multi-Level Indexing for Distributed Trajectory Streams

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

Abdul Samad

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

Dr. Eleazar Leal

May 2021

Acknowledgements

I would like to thank my advisor, Dr. Leal for making this work possible and his valuable mentorship throughout the program. He has been kind enough to take time out of his busy schedule and to guide me through different aspects of my academic and professional endeavours. I am also grateful to all of my teachers for the opportunities and encouragements to learn and succeed.

Since most of our work does not always reflect the contributions of our family and friends who have a huge role in keeping us motivated, I would like to specifically mention the support of my wife, Areeha. She has been my partner in life, work, and this academic journey which otherwise would have been strenuous. I am also fortunate enough to receive dear wishes from our siblings and parents.

Dedication

I attribute this work and my achievements in general to my late mother, Meena Tariq. She was a hardworking woman, taught us the same and shaped me into a person that I am today.

Abstract

The popularity of location-based social media and GPS-enabled mobile devices has produced a large amount of streaming trajectory data. Each streaming trajectory consists of the sequence of positions that a moving object occupies in time and is generated in an online fashion, coming at high speed. Disciplines such as social networking, urban planning, ecology, and epidemiology have great interest in querying this type of data. However, the large volume of streaming trajectories poses scalability challenges that can be addressed by efficient indexing structures and in-memory distributed architectures such as Spark. Despite this, no streaming trajectory query processing algorithm has been proposed that uses indices and distributed architectures to tackle this large-scale problem.

To address this, **we propose a novel in-memory predictive multi-level indexing technique, called PIMMLI, that leverages the distributed Spark Streaming framework to process spatio-temporal queries on streaming trajectories in an efficient manner.** We evaluated the effectiveness of PIMMLI on 3 real-life large-scale datasets. These experiments showed that PIMMLI had an average improvement of **3.5X** in total query execution and indexing time over DITA, an existing state-of-the-art batch processing algorithm for spatio-temporal querying on trajectories, and of **34.09X** in query execution time over an approach that uses no indices.

Table of Contents

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**

1.1 Background 1

1.1.1 Trajectories 1

1.1.2 Range Queries 4

1.1.3 Distributed Computing Frameworks 5

1.2 Objectives 10

1.3 Related Work 11

1.3.1 Non-Streaming Serial Spatio-Temporal Algorithms 11

1.3.2 Streaming Serial Spatio-Temporal Algorithms 12

1.3.3 Non-Streaming Distributed Spatio-Temporal Algorithms . . . 12

1.3.4 Streaming Distributed Spatio-Temporal Algorithms 14

1.3.5 Summary 14

1.4 Contributions 16

1.5 Outline 16

2 PIMMLI: Predictive In-Memory Multi-Level Indexing **17**

2.1 Overview 17

2.2 Algorithm Description 18

2.2.1 Phase 1: Pre-processing 18

2.2.2 Phase 2: Trajectory Model Training 22

2.2.3 Phase 3: Future Point Prediction 22

2.2.4 Phase 4: Multi-Level Indexing 23

2.2.5 Phase 5: Model Validation for Future Batches 27

3 Experimental Evaluation **29**

3.1 Experimental Setup 29

3.2 Performance Analysis 30

3.2.1 Datasets 30

3.2.2 Hardware 33

3.2.3 Competing Techniques 35

3.2.4 Evaluation Metrics 37

3.2.5 Parameters 38

3.3 Experimental Results 41

3.3.1 Impact of Datasets 41

3.3.2 Impact of Trajectory Counts 43

3.3.3 Impact of Hardware 44

3.3.4 Impact of Window Size 46

3.3.5 Impact of Future Point Count 47

3.3.6 Impact of Validation Error Threshold 52

3.3.7 Impact of Training Interval 55

4 Conclusions & Future Work **58**

4.1 Conclusions 58

4.2 Future Work 60

References **62**

List of Tables

1.1 Existing techniques and challenges 15

3.1 Experiment parameters 39

List of Figures

1.1 Example of a spatio-temporal trajectory. The differences between the values of the points in the trajectory are highlighted in bold. 2

1.2 Example of a range query (RQ) around the RQ point with radius ε. Trajectories with yellow points are the results of the query 4

1.3 Performance of Hadoop and Spark for a logistic regression task on a 50-node cluster. Image taken from [98] 7

1.4 Apache Spark architecture containing worker nodes managed by cluster manager for driver program 8

1.5 Discretization of input data streams in Apache Spark Streaming . . . 9

2.1 Stream processing of n trajectories 21

2.2 Example of a future point prediction 23

2.3 Example of MBR grouping 24

2.4 Partitioning using first and last points 25

2.5 Global Index R-tree using MBRs of first points 26

2.6 Global Index R-tree using MBRs of last points 27

2.7 Local Index using MBRs of first, last and pivot points 28

3.1 Sample of the GeoLife dataset trajectories mapped over Beijing, China 31

3.2 Sample of the Porto taxi dataset trajectories mapped over Porto, Portugal 33
3.3 Sample of the Beijing taxi dataset trajectories mapped over Beijing, China 34
3.4 Total execution time using the naive approach 36
3.5 Cumulative execution time using the naive approach 37
3.6 Total execution time using DITA in streaming 38
3.7 Cumulative execution time using DITA in streaming 39
3.8 Performance over the Beijing taxi dataset 42
3.9 Performance over the Porto taxi dataset 42
3.10 Performance over the GeoLife dataset 43
3.11 Total impact of trajectories count 44
3.12 Cumulative impact of trajectories count 45
3.13 Performance at Janus 46
3.14 Performance at Ukko 46
3.15 Total impact of window size (w) 48
3.16 Cumulative impact of window size (w) 49
3.17 Total impact of future point count (f) 50
3.18 Cumulative impact of future point count (f) 51
3.19 Total impact of validation error threshold (ϵ) 53
3.20 Cumulative impact of validation error threshold (ϵ) 54
3.21 Total impact of training interval (t) 56
3.22 Cumulative impact of training interval (t) 57

1 Introduction

In this chapter, we introduce the problem of indexing in data streams. In section 1.1 we introduce terminologies such as trajectories and range queries. We provide formal definitions and give examples of types of queries issued to spatio-temporal data and explain their significance. We also introduce distributed computing in the context of *Apache Spark* and discuss its research challenges. In section 1.2 we state our objective for this thesis and research questions that we aim to answer through our work. Section 1.3 presents related work. We list all the contributions that this research makes in section 1.4. To help navigate all parts of this thesis, we also provide an outline in section 1.5 later in this chapter.

1.1 Background

In this section we provide formal definitions of terms used in this thesis. This includes spatio-temporal trajectories in subsection 1.1.1 and range query in 1.1.2. We also discuss the research challenges in distributed computing with the frameworks we used in subsection 1.1.3.

1.1.1 Trajectories

In this research, we deal with a special type of data called *trajectory* data. Each trajectory is a sequence of objects moving in space with respect to time. Such data can be generated by location sensors such as cellphones containing GPS receivers to determine the location of a device [30]. These spatial data points are also associated with timestamps making it a spatio-temporal sequence. Points in trajectories usually also contain other non-spatial attributes [26] such as device status, user ids, soil status [50], etc.

Figure 1.1 shows an example of a trajectory of a taxi driving in Beijing city, a trajectory that was taken from the GeoLife [104] dataset. The first point was collected at location (latitude: 39.97555, longitude: 116.3308) at time 12:31:00. After two seconds, at time 12:31:02, the object moved to the location (39.97557, 116.3308). Similarly, the point moves ahead and the path that the object takes through these points forms a trajectory.

As defined earlier, spatio-temporal trajectories are a discrete sequence of spatial and temporal coordinates which may also contain other attributes. For a point i in a two dimensional space with coordinates (x_i, y_i) at a time instant t_i, a *trajectory* can be formally defined as the set $\{(x_i, y_i, t_i) \in \mathbb{R}^3 \mid t_i \leq t_{i+1}, 1 \leq i \leq n\}$, where i is an integer.

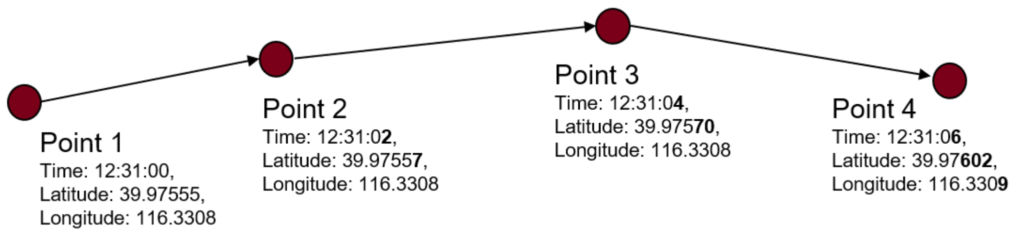


Figure 1.1: Example of a spatio-temporal trajectory. The differences between the values of the points in the trajectory are highlighted in bold.

To facilitate faster query processing, we build indexes which are comprised of data structures to navigate through the available data. During a search in possibly huge amounts of data, these structures help in locating the relevant segments. By finding the appropriate segments, candidates can be effectively filtered and refined for a shorter query execution times. An example of such data structure is binary search tree [7] which can reduce the search time of an ordered set of elements from linear to logarithmic time.

Due to the complexity of multi-attribute values in a trajectory, this type of data requires special data structures and algorithms for efficient storage [18, 105, 11, 17, 39, 63, 61], indexing [89, 35, 71, 13, 49] and querying [34, 36, 80, 75, 83].

Conventionally, *batch-oriented systems* have been used to efficiently store and index trajectory data [71, 74, 10, 90]. These systems require that the entire dataset is stored in RAM or disk throughout the analysis to build indexes and process queries.

The popularity of huge sources of such trajectory data [57, 103] introduced another type of systems which are stream-oriented, i.e. systems that process *data-streams* [2, 62]. Formally, *trajectory data streams* refers to the type of trajectory data that evolves in an online fashion, and contains the instantaneous position of the object as well as previous points that can be cumulatively used to analyze behaviors over time [19]. An example of such trajectory data-stream can be depicted by figure 1.1, where the last point, i.e. Point 4, represents the current position of the object and is recently added to stream, whereas previous points, i.e. Points 1,2,3, were previously streamed and are present in memory to analyze the evolving behavior of a trajectory.

To avoid exhaustive search and then reduce query processing times, stream-oriented systems require indexing the incoming data in an online fashion as data is received at high speed [4, 43, 22, 8]. These systems introduce several challenges to indexing as they require complete data at once [28, 32].

Realizing the potential and importance of huge applications, there are several large real-life datasets containing such trajectories. An example of a huge collection was published by Microsoft Research Asia containing 17,000+ trajectories of outdoor human activities in the GeoLife dataset [104]. In 2015, a Taxi Trajectory Prediction challenge at Kaggle [1] also published a dataset containing 10,000+ spatio-temporal trajectories of taxi rides in the city of Porto, Portugal [60].

1.1.2 Range Queries

The significance of efficiently indexing such trajectory data-streams comes from the applications that require frequent issuance of queries. One type of query on spatio-temporal data is the *range query* [66]. As demonstrated by figure 1.2, a range query for spatial data is issued in terms of a geolocation point, named RQ in the figure, and a spatial range threshold called $\varepsilon \in \mathbb{R}^+$. This query returns the trajectories having all their points located within that range. An example application of this query is in ride-hailing services such as Uber [101], Lyft etc. with frequent queries like "finding all drivers within 2 miles of a user's location".

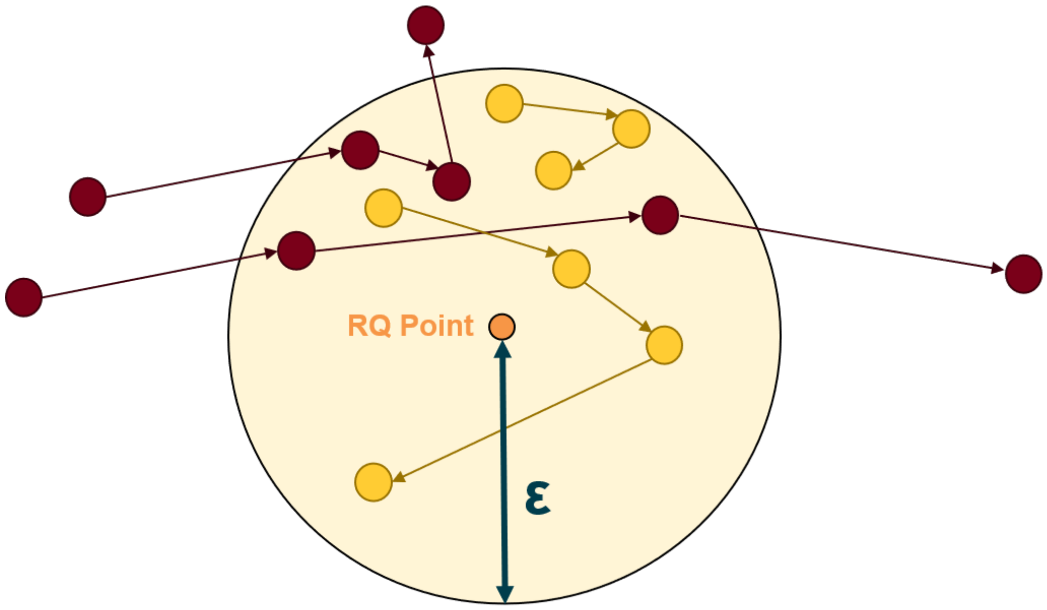


Figure 1.2: Example of a range query (RQ) around the RQ point with radius ε. Trajectories with yellow points are the results of the query

[1]https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i

Given a trajectory dataset D with Q set of points within each trajectory, a spatial point p, a threshold $\varepsilon \in \mathbb{R}^+$, a range query $r(D, p, \varepsilon)$ returns all trajectories containing points $q \in Q$ such that $dist(p, q) \leq \varepsilon$ where $dist(p, q)$ is the *Euclidean distance* between points p and q.

The essence of work in this research is to be able to efficiently perform range queries, which find applications in urban planning [92], location-based social networks [93, 16] and ecology [40], where ecologists estimate animal migration routes which are essential to understanding population dynamics. The range query is one of the most popular and common types of queries issued to spatial data [81, 64, 91].

By nature, range queries require the collection of results spanned over a possibly large area, and are thus I/O intensive [68]. When executing range queries, a large number of trajectories may be accessed which may be supplied through several partitions at a time, therefore, the *I/O complexity* is particularly important when referring to partitions accessed in the index structure [45]. To address this challenge, packed [47] R-trees [33] with partitions containing *minimum bounding rectangles* (MBRs) of spatial data are used and are one of the most common ways to efficiently perform range queries [46]. However, these techniques suffer from additional computational overhead when maintaining index structures of continuously changing data.

1.1.3 Distributed Computing Frameworks

Because of the ubiquity of mobile devices, the data collected for each user has increased at an unprecedented rate [77]. To cope with such an abundant amount of data, modern spatial data systems need to utilize *distributed systems* [79, 73]. These systems enable store and compute over a cluster of different machines and allow combining inexpensive commodity hardware to build powerful systems [102].

However, these come with additional challenges of *data partitioning* and *transfer costs*. *Data partitioning* consists in evenly dividing the computational tasks across multiple nodes with the objective of reducing query execution time and is required to efficiently distribute data across the cluster of nodes such that queries need to access least nodes possible. We also need to ensure that the data transferred between nodes while computing any intermediate results is minimal, reducing *transfer costs*. This is because *transfer costs* involve communication over a network, which is significantly larger than CPU computation time.

Since issuing range queries on huge amounts of spatial data is computationally expensive [88, 41], our work is performed over distributed systems using the Apache Spark framework [97], which is an in-memory platform and has shown an average of 10X better performance in iterative computation than other frameworks such as Hadoop [98, 58, 72]. For the purposes of stream management using a framework within our algorithm, we have incorporated Spark Streaming[2], which is able to handle higher maximum throughput [15] than other popular platforms such as Apache Flink [12] and Storm [44]. Here, we will discuss these platforms in more detail and discuss the challenges that this research faces in using these systems.

Apache Spark

Apache Spark is an in-memory distributed computing platform, as opposed to Hadoop [76], which is disk-based and a predecessor in distributed computing platforms. We chose Apache Spark for the following reasons:

- Spark maximizes memory usage for intermediate operations making it much faster [97] than Hadoop, which is a disk-based framework and performs data flushes and reloads from disk for intermediate results.

[2]https://spark.apache.org/streaming/

- Spark can solve iterative problems and graph algorithms much faster due to its global cache mechanisms [31]. It can also efficiently cache results from previous queries [37]. Figure 1.3 shows a performance comparison between Hadoop and Spark in terms of the running time to fit a logistic regression model [98], which requires an iterative algorithm to optimize a mathematical function using gradient descent. Our research builds trajectory models which are trained in an iterative manner, and thus our algorithm leverages the iterative performance advantage of Spark.

- Spark is fault-tolerant and can recover from node and partition failures [97]. An important abstraction of data in Spark are *Resilient Distributed Datasets (RDDs)* which are data structures for fault-tolerant data-replication [99]. It uses an approach called *lineage* in which RDDs keep track of graph of transformations that happen to a base dataset and recover data through those transformations.

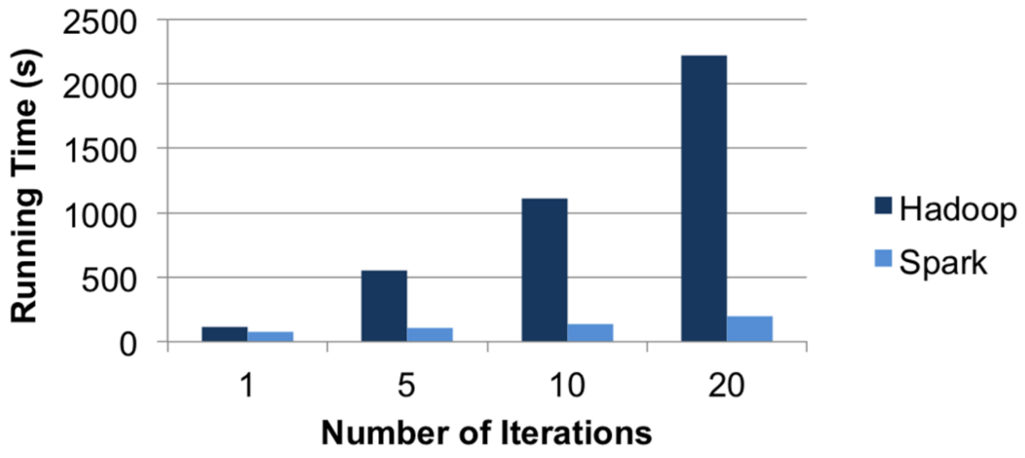


Figure 1.3: Performance of Hadoop and Spark for a logistic regression task on a 50-node cluster. Image taken from [98]

As shown in figure 1.4, an application written for Spark consists of a *driver* program and *executors* on a cluster of *worker* nodes. Partitions for distributed storage are built across the worker nodes and also have a common *shared* memory. A *cluster manager* between driver and worker nodes allows distribution and management of resources and tasks. Spark also provides a *standalone* cluster manager which can be replaced by any other resource managers such as Mesos [38] and YARN [84]. Spark also supports running applications faster in Hadoop Cluster by reducing reads and writes to disk for intermediate results [1].

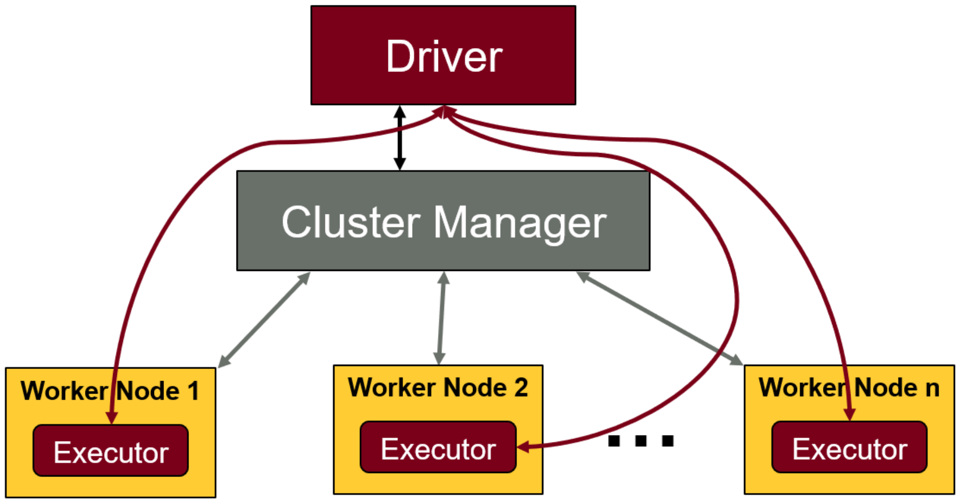


Figure 1.4: Apache Spark architecture containing worker nodes managed by cluster manager for driver program

Spark Streaming

Built on top of the Apache Spark platform, Spark Streaming offers high throughput stream processing in distributed systems. In Spark Streaming, data is divided into small *Discretized Streams (DStreams)* which are essentially mini-batches of streaming data. This allows for a relatively easier transfer of batch-oriented algorithms to streaming platform [9]. Figure 1.5 shows an example of discretization of an input data stream. In this example, a continuous data stream is split into batches labelled

1 through n and the complete set of such discrete batches form a *DStream*.

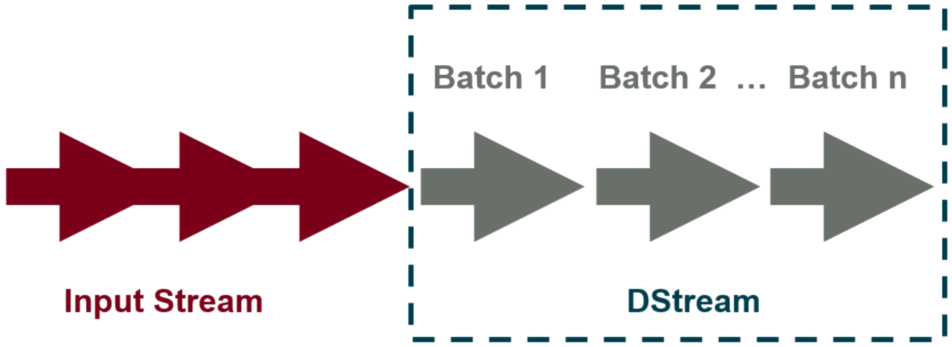


Figure 1.5: Discretization of input data streams in Apache Spark Streaming

Research Challenges of Spark

Here we mention the challenges that this research faces in utilizing Apache Spark and the Spark Streaming framework:

- Apache Spark performs much faster than Hadoop by utilizing RAM for most computations [97]. However, this also presents a challenge of high memory requirement as Spark's speed advantage is reduced in memory-constrained environments. Each worker node needs to have sufficient memory to efficiently run an algorithm completely in the RAM of an executor, otherwise it would need to flush intermediate results to disk. This is critical for a use-case where the data on each worker node could be too large to fit in memory. In our case, this means that possibly very large trajectories stored in partitions of worker nodes cannot be accessed completely in memory at a time. One of the ways to overcome this challenge is by building in each partition local indexes that can be light-weight representations of actual trajectories and can fit in memory.

This ensures maximum in-memory computation at each worker node without accessing complete data.

- Although Spark Streaming provides a higher maximum throughput capability than other streaming frameworks, it has a much higher latency especially at higher throughputs [15] which may be an important factor in choosing a streaming framework for some systems in production. The latency in Spark Streaming is sensitive to the batch duration, which is an application-dependent parameter that needs to be determined by trial-and-error and requires an optimum value to be set depending on each application [56].

- The challenge produced by the streaming nature of the data is the requirement of frequent index updates which can add additional computational overhead. The query speedup provided by the indexing is attenuated by the cost of indexing each batch of the stream. As Spark discretizes the stream, each batch adds limited points, which need to be immediately indexed to maintain the correctness of query results. This trade-off between correctness and computational expense can be critical in applications like ride-hailing services where a client application may be actively querying for a recently available driver [101]. In such application, it is essential for a query to return correct results by including the recently added data points, i.e. location of recent driver.

1.2 Objectives

In this thesis, we address the challenges of indexing data-streams of complex spatio-temporal trajectories in distributed in-memory systems. Our research objective encompasses the following goals:

- Develop an algorithm to predictively index trajectories that arrive in continuous streams. Such indexing should enable issuing range queries that return correct results faster than an exhaustive search.
- Compare our approach to the existing state-of-the-art approach named DITA, in batch-oriented systems and evaluate it on the same baseline of in-memory computation of data streams.
- Thoroughly evaluate our algorithm and observe the impact of different settings by performing experiments on different hardware systems, datasets, trajectory counts and algorithm parameters.

1.3 Related Work

There are several recent works using distributed computing that deal with spatial or spatio-temporal trajectory data. Most of them use similarity [100] and join queries [95, 74], whereas very few address the challenges of range queries [10]. Here we split the works in sub-sections of Serial Spatio-Temporal Algorithms (non-streaming), Serial Spatio-Temporal Algorithms with Streaming, and Distributed Spatio-Temporal Algorithms with and without streaming.

1.3.1 Non-Streaming Serial Spatio-Temporal Algorithms

One of the earliest works in the area of non-streaming serial spatio-temporal algorithms [82], introduced the idea of indexing and issuing similarity search queries for spatial data. This work presented a novel idea of two-phase divide-and-conquer indexing technique to prune search space.

Similarly, TrajTree [71] presented an index structure optimized for kNNs queries on spatio-temporal trajectory data. SETI [13] used two-level index structures to separately deal with spatial and temporal components of trajectories. Also, [6] and [5] demonstrated trajectory joins using serial algorithms to prune trajectories.

SharkDB [87] presented an in-memory indexing structure for serial query algorithms on trajectory data. This work focuses on a frame structure index optimized to fit in limited memory. SharkDB demonstrated the usage of these index structures over kNN and window-based queries.

However, these approaches can neither be scaled to distributed systems nor can they incorporate streaming framework as these works are highly optimized for serial execution in batch-oriented settings.

1.3.2 Streaming Serial Spatio-Temporal Algorithms

PLACE [59] introduced the notion of indexing and query trajectory data-streams in-place by extending the processing of continuous sliding window queries. [29] presented a framework to deal with spatio-temporal trajectory data streams in an abstract manner. A more practical approach in terms of road networks is presented in [67] which also deals with streaming data. However, in all of these works the algorithms used are serial and cannot be directly extended to distributed systems.

1.3.3 Non-Streaming Distributed Spatio-Temporal Algorithms

DITIR [10] processes range queries and distributed indexing using H-Base, a Hadoop based system [86]. It uses B+ trees for indexing which is a good choice for index update in high throughput data insertion, but it is less efficient for indexing spatial data than R-trees [33]. Also, Hadoop is a disk-based framework, in contrast to Apache Spark, that efficiently leverages memory for maximum computations and caching intermediate results. There are other similar works that use H-Base for trajectory queries including DFTHR [70], NODA [48] and others [53]. There are several works [24, 78, 54, 55, 25] that are built over Hadoop and extend its functionality to support spatio-temporal data for indexing and querying.

ST-Hadoop [3] introduced a Map-Reduce framework for multi-level indexing in spatio-temporal trajectories by building temporal and spatial indexes separately. It supports range, kNN, and join queries. Similarly, GeoSpark [94] supports the same queries (range, kNN, and join) and is built using Apache Spark framework but it uses only spatial data. This technique uses R-tree and Quad-tree as its index structures.

SIMBA [90] also performed distributed indexing and uses Apache Spark, however it has a single index which cannot be efficiently scaled to several partitions when trajectories are stored in a distributed environment.

DITA [74] presented distributed indexing for spatial trajectory data and also several similarity query and join query experiments against existing systems like SIMBA [90].

DISON [95] uses distributed indexing similar to DITA and defines the problem in terms of road networks. DISON introduces a road-network-aware trajectory similarity function and uses local inverted partitions.

In most recent works, TrajMesa [52] uses Z-order curve indexing for spatio-temporal trajectories. The work is based on the GeoMesa [42] framework, which is an indexing toolkit for spatio-temporal data built on top of Hadoop and H-Base. TrajMesa demonstrates application of several types of queries including kNNs, spatial range and similarity query. UlTraMan [21] offers similar queries, uses R-tree-based indexing and is built on top of Apache Spark.

However, all of these works address indexing challenges in batch-oriented systems.

Our work focuses on the constant evolution of such data, i.e. data stream management systems, where resource-intensive continuous queries present a unique challenge.

1.3.4 Streaming Distributed Spatio-Temporal Algorithms

TraSP [65] presents online trajectory similarity processing. It deals with streams of spatial trajectory data; however, it does not propose any indexing methods and instead uses matrix-based partitioning with a greater *transfer cost*.

DCS [20] presents an approach for top-K trajectory similarity processing in real-time environments. It uses the *longest common subsequence (LCSS)* as a distance measure to perform space-based partitioning.

A very recent study, Dragoon [27] uses a mutable RDD model for online update of RDDs in streaming and non-streaming frameworks. It uses historical data for index construction and does not use any predictive indexing techniques for trajectory streams. Similarly, a technique [14] processes clustering and co-movement pattern detection over trajectory streams.

To the best of our knowledge, there exists no efficient distributed indexing and range query processing algorithm for streams. Our work also incorporates building these complex index structures predictively to address the challenges of frequent index updates.

1.3.5 Summary

We present a summary of related works in the table 1.1. It lists some of the existing techniques with their addressed challenges. The supported queries column lists queries such as kNN, range, similarity and join supported by the indexes of that technique. The Streaming Support specifies whether a technique supports trajectory

Technique	**Supported Queries**	**Streaming Support**	**Distributed Computing**	**Offline Indexing**	**Predictive Indexing**
TrajTree [71]	kNN	✗	✗	✓	✗
TrajStore [18]	Range	✗	✗	✓	✗
ST Hadoop [3]	Range, kNN, Join	✗	✓	✓	✗
DITA [74]	Similarity, Join, Range	✗	✓	✓	✗
DITA extended	Similarity, Join, Range	✓	✓	✓	✗
DITIR [10]	Range	✗	✓	✓	✗
TraSP [65]	Similarity	✓	✓	✗	✗
UlTraMan [21]	Range, kNN	✗	✓	✓	✗
Dragoon [27]	Range, kNN	✓	✓	✓	✗
PIMMLI (this work)	Range	✓	✓	✗	✓

Table 1.1: Existing techniques and challenges

streams as its input for indexing and querying. The Distributed Computing column is checked if the technique uses any distributed computing frameworks such as Hadoop or Spark. The Offline Indexing is marked if it supports batch-oriented processing. Also, Predictive Indexing is marked for techniques that predictively build indexes to maximize index reusability in streams. As presented in the table there is no technique except our approach, PIMMLI that uses predictive indexing along with support for streaming and distributed computing when executing range queries.

1.4 Contributions

In this research, we make following contributions:

- We present PIMMLI, the first algorithm to predictively index distributed trajectory data streams. It performs indexing and querying 3.5X faster than DITA [74], an existing state-of-the-art algorithm for batch-oriented processing, and 34.09X faster than the naive approach.
- We extend a state-of-the-art batch-oriented system, DITA, to work with trajectory streams and use it as a baseline against our algorithm.
- We evaluate PIMMLI against two competing approaches on 3 real-life datasets of different characteristics and sizes: GeoLife [104], Porto [60], and Beijing Taxi [96] datasets. We perform experiments demonstrating the impact of several parameters, hardware systems and trajectories on the performance of PIMMLI.

1.5 Outline

The remaining chapters of this thesis are organized as follows. Chapter 2 introduces our proposed algorithm, PIMMLI, and details its different phases. It also includes the description of all important variables in performance analysis, such as datasets, hardware systems, competing techniques, evaluation metrics and algorithm parameters. Chapter 3 presents experimental results. It demonstrates the impact of the different variables we described in performance analysis of Chapter 2. Lastly, we present Conclusions and Future Work in Chapter 4.

2 PIMMLI: Predictive In-Memory Multi-Level Indexing

In this chapter, we present our proposed algorithm, PIMMLI. We start by presenting an overview in section 2.1 and walk through the pseudo-code. Section 2.2 provides details on each stage of our algorithm and explains each phase with an example.

2.1 Overview

PIMMLI is an algorithm to predictively index data-streams of spatio-temporal trajectories. It foresees future data points and builds multi-level indexes on such data. By proceeding in this manner, PIMMLI achieves shorter query execution time without paying the high computation cost of indexing the entire dataset every time a data-point arrives from one of the stream.

PIMMLI is built on top of the Apache Spark engine that runs on a discretized stream, where it processes each incoming batch in a window of defined size. There are five major stages of our algorithm. Our first phase consists of discretization and processing of each batch. In the second phase, we train the models for representing trajectories. Each trajectory owns a separate model which is initially trained with incoming batches of points. The third phase predicts the future points for each trajectory. In our fourth stage, we make use of existing and predicted points for each trajectory to build indexes. To address the challenge of memory constraints in Spark, multi-level indexing is done to avoid accessing complete trajectories even in worker nodes. Local indexes are built in each worker node's partitions in addition to global index in shared memory. In our final phase, which is executed with every subsequent batch, we validate our model for each trajectory. If the average error of trajectory models grows over a certain predefined threshold, we re-train the models and update the indexes.

The pseudo-code of PIMMLI is split into algorithms 1, 2 and 3. Algorithm 1 demonstrates the first phase of PIMMLI, where each incoming batch in stream is discretely processed. It serves as the entry point for all batches and directs them to relevant sub-algorithms. Algorithm 2 presents all functions related to model training, prediction and validation, therefore, corresponding to Phase two, three and five respectively. Algorithm 3 corresponds to Phase four of PIMMLI where we perform multi-level indexing of trajectories. Each function in these pseudo-codes is explained in section 2.2 with a detailed walk-through for each phase.

2.2 Algorithm Description

This section explains each phase of PIMMLI in separate sub-sections with examples. The details also walk through the pseudo-code and provide explanations for each function separately.

2.2.1 Phase 1: Pre-processing

A continuous data stream is divided into batches and is termed as discretized stream or DStream. Internally in Apache Spark, each batch in a DStream is a sequence of RDDs (Resilient Distributed Datasets) [1] which are immutable, partitioned

[1]https://spark.apache.org/docs/latest/api/scala/org/apache/spark/rdd/RDD.html

Algorithm 1 Predictive Indexing for Trajectory Data Streams

Require: db: Database of Previously Streamed Trajectories, w: Window Size, b: Newest Batch in Stream, i: Batch Index, t: Training Interval, ϵ: Validation Error Threshold, f: No. of Points to Predict

Ensure: $|db| \geq w$

```
 1: function ProcessBatch($db, b, i, w, t, \epsilon, f$)
 2:     $db \leftarrow db \bigcup b$
 3:     Initialize the $models$ array
 4:     if ($i$ is 1 or a multiple of $t$) or AvgValidationError($db, b, models$) $\geq \epsilon$ then
 5:         $models \leftarrow$ Train($db, models$)
 6:         Initialize the $updatedTrajectories$ array
 7:         for each $trajectory$ in $db$ do
 8:             $futurePoints \leftarrow$ Predict($models[trajectory]$, $trajectory$, $f$)
 9:             $updatedTrajectories[trajectory] \leftarrow trajectory \bigcup futurePoints$
10:         end for
11:         Indexes $\leftarrow$ BuildIndexes($updatedTrajectories$)
12:     end if
13:     Queries search in $db$ using $Indexes$
14: end function
```

collections of objects computable in a distributed setting. A window of defined size w is slided over such DStream allowing a number of a batches to be considered together for developing or updating internal representation of each trajectory. An example of a stream processing is demonstrated for n trajectories with $d + w + 1$ points in figure 2.1. For any trajectory, the earliest points p_1, p_2, ..., p_d are stored in a database of size d. As new points arrive, such as p_{d+w+1} in this example, they are appended to the window of size w. The window itself contains points p_{d+1}, p_{d+2}, ..., p_{d+w} which are processed for model training or validation in the current batch. For n trajectories, the database and window sizes become $d \cdot n$ and $w \cdot n$ respectively. This stage corresponds to the ProcessBatch function in the pseudo-code of algorithm 1. The function accepts the incoming batch in the stream with its index, along with other arguments to be used. The database is also updated with the newest batch in the

Algorithm 2 Model Training, Prediction and Validation

```
function TRAIN($db, models$)
    for each $trajectory$ in $db$ do
        $models[trajectory] \leftarrow$ AVGDISTANDANGLE($trajectory$)
    end for
    return $models$
end function

function PREDICT($model, trajectory, f$)
    $lastPoint \leftarrow trajectory.lastPoint$
    Initalize $points$ array
    for $i \leftarrow 1$ to $f$ do
        $points[i] \leftarrow$ Move $lastPoint$ by $avgDistance$ and $avgAngle$ in $model$
        $lastPoint \leftarrow points[i]$
    end for
    return $points$
end function

function AVGVALIDATIONERROR($db, b, models$)
    for each $trajectory$ in $db$ do
        $avgHeading \leftarrow models[trajectory]$
        $newHeading \leftarrow$ AVGDISTANDANGLE($b[trajectory]$)
        $errors[trajectory] \leftarrow$ EUCLIDEANDISTANCE($avgHeading, newHeading$)
    end for
    return Average of $errors$
end function

function AVGDISTANDANGLE($trajectory$)
    Initalize $distances$ and $angles$ arrays
    $previousPoint \leftarrow trajectory.firstPoint$
    for each $point$ in $trajectory - previousPoint$ do
        Add EUCLIDEANDISTANCE($previousPoint, point$) to $distances$
        Add ANGLE($previousPoint, point$) to $angles$
        $previousPoint \leftarrow point$
    end for
    $heading \leftarrow$ Average of $distances$, Average of $angles$
    return $heading$
end function
```

Algorithm 3 Multi-Level Indexing

```
1: function BuildIndexes($trajectories$)
2:     Group closest first points of $trajectories$
3:     for each $Group$ do
4:         Group last points
5:         Each sub-group forms a partition
6:     end for
7:     MBR$_F$← MBRs of groups of first points
8:     MBR$_L$← MBRs of groups of last points
9:     $GlobalIndex \leftarrow RTree(MBR_F), RTree(MBR_L)$
10:    for each $partition$ do
11:        $LocalIndex[partition] \leftarrow$ Build a Trie-like structure of first, last and pivot points
12:    end for
13:    return $GlobalIndex, LocalIndex$
14: end function
```

stream to be used for complete indexing if the model is re-trained.

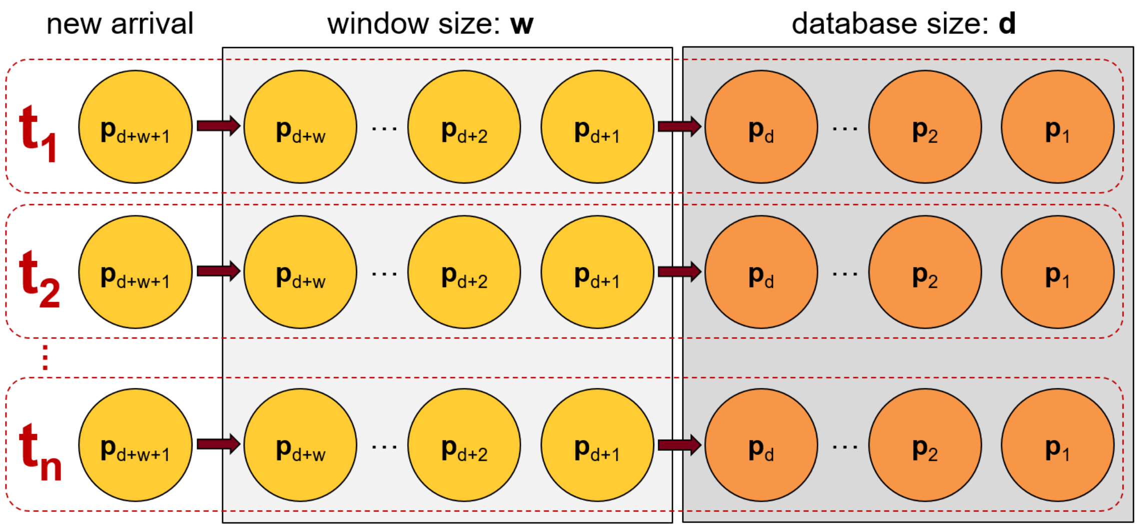


Figure 2.1: Stream processing of n trajectories

2.2.2 Phase 2: Trajectory Model Training

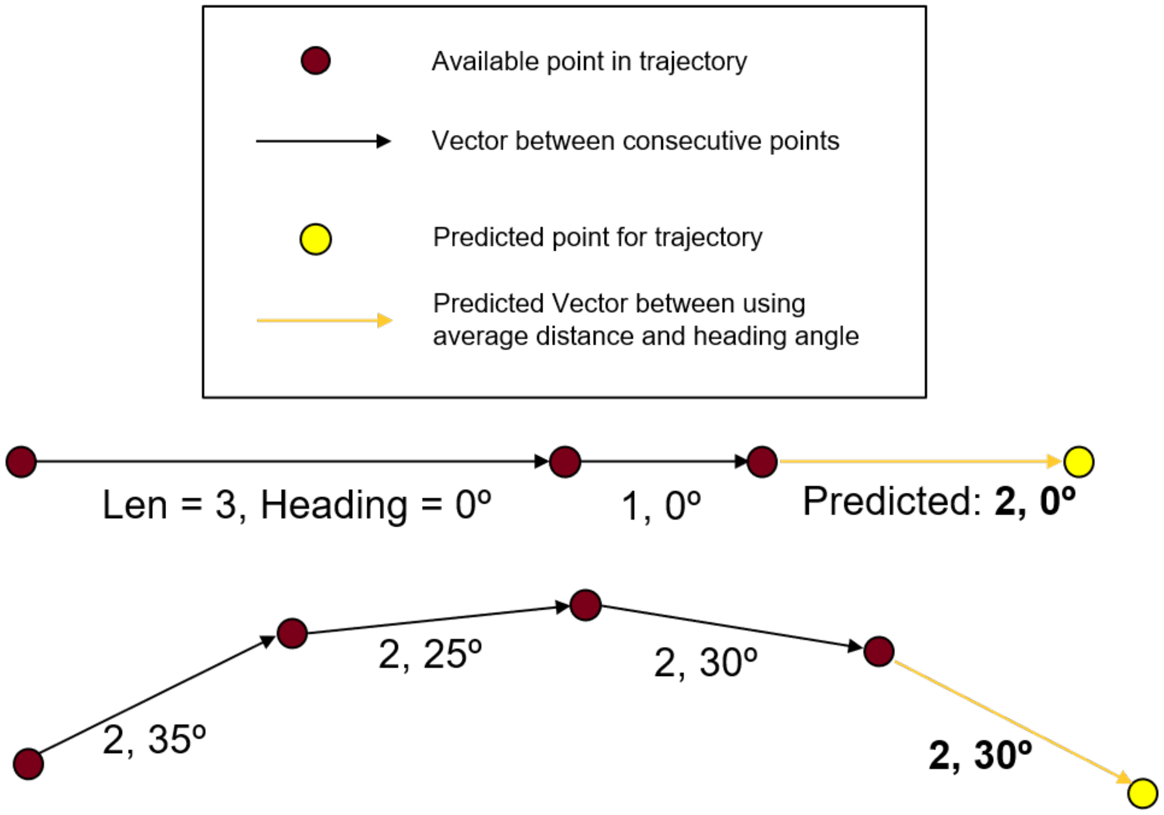
For each trajectory, we develop a model that can represent the characteristics of a trajectory using few statistics, such as average distance and angle between points. Our proposed model keeps track of statistics between each pair of consecutive points of every trajectory, as indicated in the example in figure 2.2. The average distance between points in a two-dimensional space can be calculated by the average of the *Euclidean distance* between the subsequent points. For points $p_1, p_2, p_3, ..., p_n$, the $averageDistance$ is the average of $dist(p_1, p_2)$, $dist(p_2, p_3)$, ..., and $dist(p_{n-1}, p_n)$ where $dist(a, b)$ is the *Euclidean distance* between points a and b. Similarly, the $avgAngle$ is the average of $tan^{-1}\left(\frac{p_2.y \ - \ p_1.y}{p_2.x \ - \ p_1.x}\right)$, ..., and $tan^{-1}\left(\frac{p_n.y \ - \ p_{n-1}.y}{p_n.x \ - \ p_{n-1}.x}\right)$.

These averages of distances and angles between consecutive points of a trajectory are calculated in the function AvgDistAndAngle at line 27 of algorithm 2 and are returned together as *heading* in the pseudo-code. The advantage of this proposed model is that it allows easy visualization of the internal representation of trajectories and also supports online model updates, which is processing queries on streaming data.

2.2.3 Phase 3: Future Point Prediction

The model developed in the second stage and that is fit to every trajectory is then used to predict the future points of each trajectory based on the average distance and angle as demonstrated in figure 2.2. For each trajectory, these points can be combined with its existing points (streamed so far) to develop a larger trajectory which can be expensive at first to index but pays off its cost in long term, as shown by the experiments later in Chapter 3.

For points $p_1, p_2, p_3, ..., p_n$ already streamed, the point p_{n+1} can be predicted by displacing the last point p_n by $avgDistance$ in direction of $avgAngle$, such that its two dimensional coordinates are $(p_n.x + avgDistance \cdot \cos(avgAngle)\ ,\ p_n.y + avgDistance \cdot \sin(avgAngle))$. This calculation is done in the PREDICT function at line 8 of algorithm 2. The function takes as arguments *model*, *trajectory* and future point count (f). It returns f, the number of points predicted for the *trajectory* using the average distance and angle from the *model*.



Available point in trajectory
Vector between consecutive points
Predicted point for trajectory
Predicted Vector between using average distance and heading angle
Len = 3, Heading = 0°
1, 0°
Predicted: 2, 0°
2, 35°
2, 25°
2, 30°
2, 30°

Figure 2.2: Example of a future point prediction

2.2.4 Phase 4: Multi-Level Indexing

The existing and predicted points for each trajectory are indexed using a multi-level indexing scheme. The trajectories are partitioned in a distributed storage, a global index is formed to locate such partitions and a local index is built inside each partition to locate candidates for each query. The pseudo-code of algorithm 3 corresponds to this phase of PIMMLI. As indexing comprises of multiple sub-phases such as partitioning, global indexing and local indexing, we will explain them in following subsections with reference to their corresponding parts in the pseudo-code.

Partitioning

Based on the available number of nodes in the distributed system, a partition is allocated at each node. Each partition groups trajectories. For each group, the end-points of a trajectory closest to end-points of other trajectories form a *MBR* (*minimum bounding rectangle*) as shown in figure 2.3. We build groups of MBRs of first points and then in each group, we build sub-groups of MBRs of last points of trajectories. Each of these sub groups are allocated to an available partition, corresponding to line 5 of algorithm 3.

We employ Sort-Tile-Recursive (STR) partitioning technique [51] to index these MBRs in an R-tree structure. This bulk load technique ensures efficient and uniform partitioning of trajectories as compared to other spatial partitioning techniques in [23]. Figure 2.4 shows an example of partitioning of trajectories using the first and last points of the trajectories.

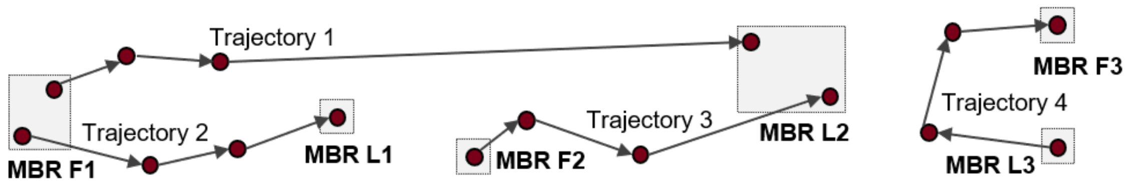


Figure 2.3: Example of MBR grouping

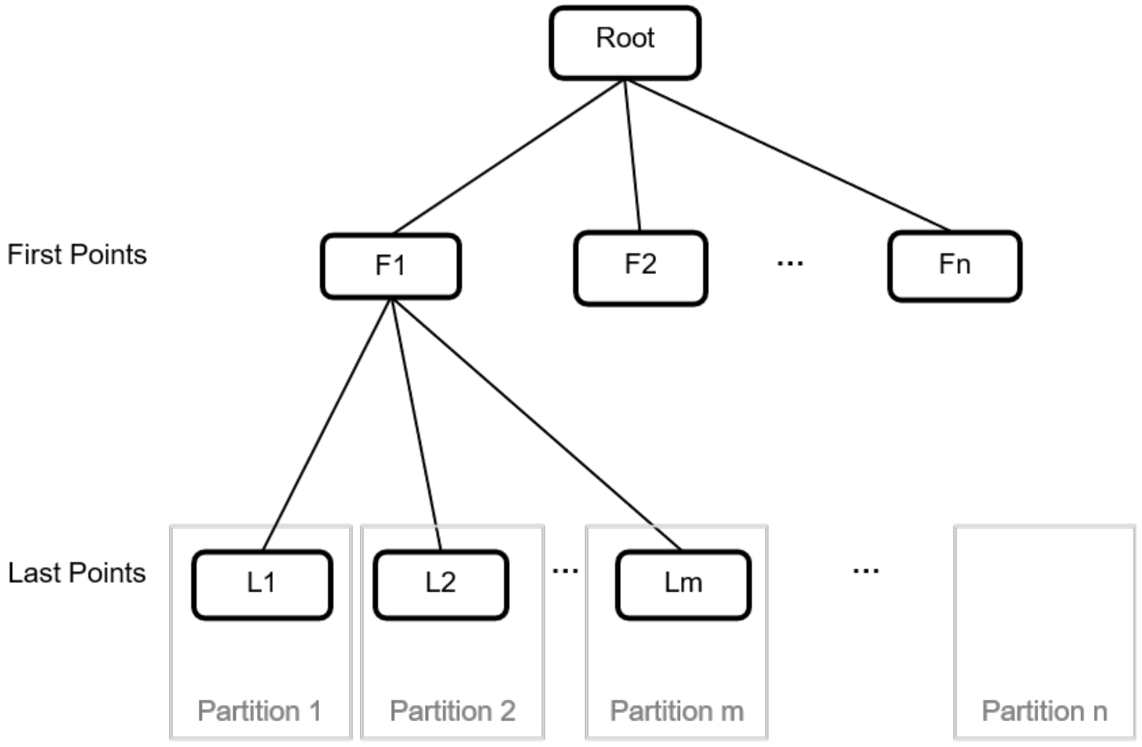


Figure 2.4: Partitioning using first and last points

Global Indexing

The Global Index helps locate the right partitions in distributed storage when processing a range of query. Since the partitions are allocated based on the first and last points of trajectories, the Global Index is comprised of two R-trees, specified by line 9 of algorithm 3. One of the R-trees loads MBRs of first points as shown in figure 2.5 and the other loads MBRs of last points as in 2.6. The Global Index is the smallest in terms of space and can be kept in memory of the Master node.

Local Indexing

The Local Indexes help locate the relevant trajectories called candidates within each partition. Candidate trajectories are lesser than the entire amount of trajectories

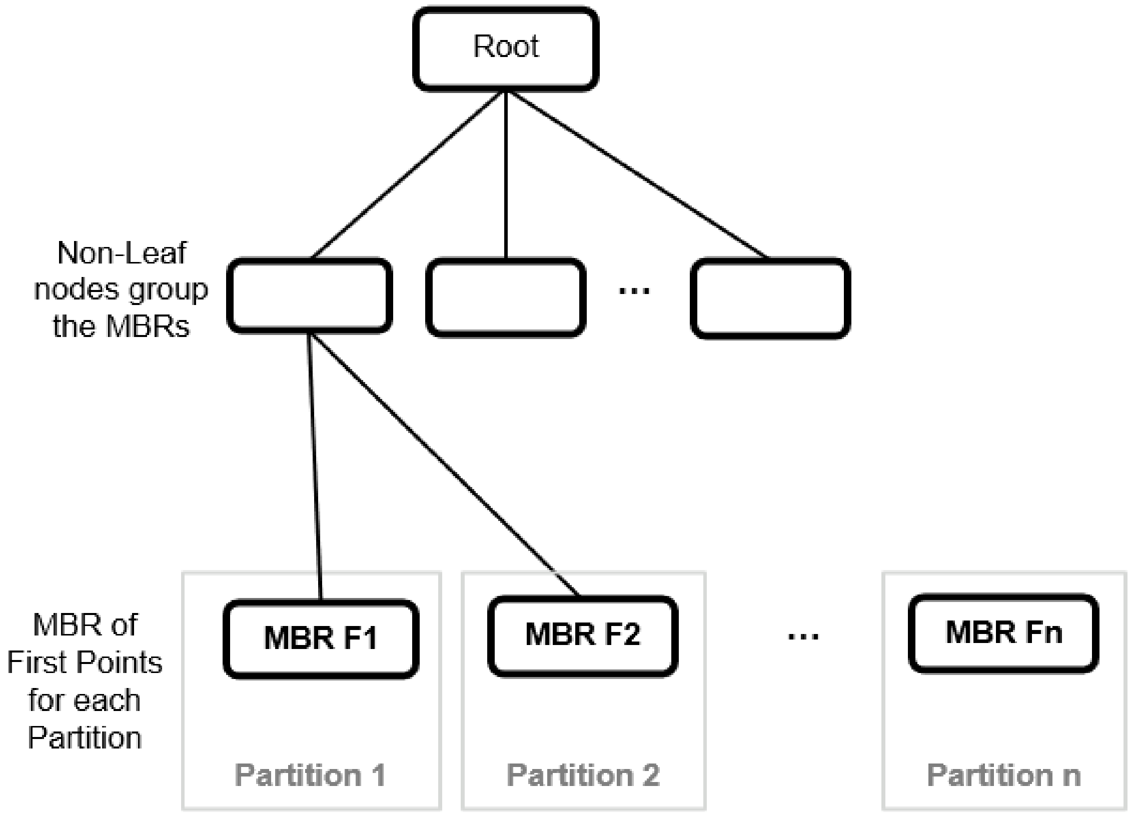

Root
Non-Leaf nodes group the MBRs
...
MBR of First Points for each Partition
MBR F1
MBR F2
...
MBR Fn
Partition 1
Partition 2
Partition n

Figure 2.5: Global Index R-tree using MBRs of first points

in the partition, yet they may contain some trajectories out of the desired range specified in our query. The Local Index is comprised of a trie-like structure to index trajectories present in that particular partition. As specified by line 11 of algorithm 3, The trie structure in our case is made of first, last and pivot points as shown in figure 2.7. The pivot points are representative points of a trajectory selected using the *neighbor distance*, which is a distance between any two consecutive points in a trajectory. A point is a pivot point among others if it has the largest neighbor distance. These allow us to approximately represent a trajectory without needing to index each point. A Local Index is present in each partition.

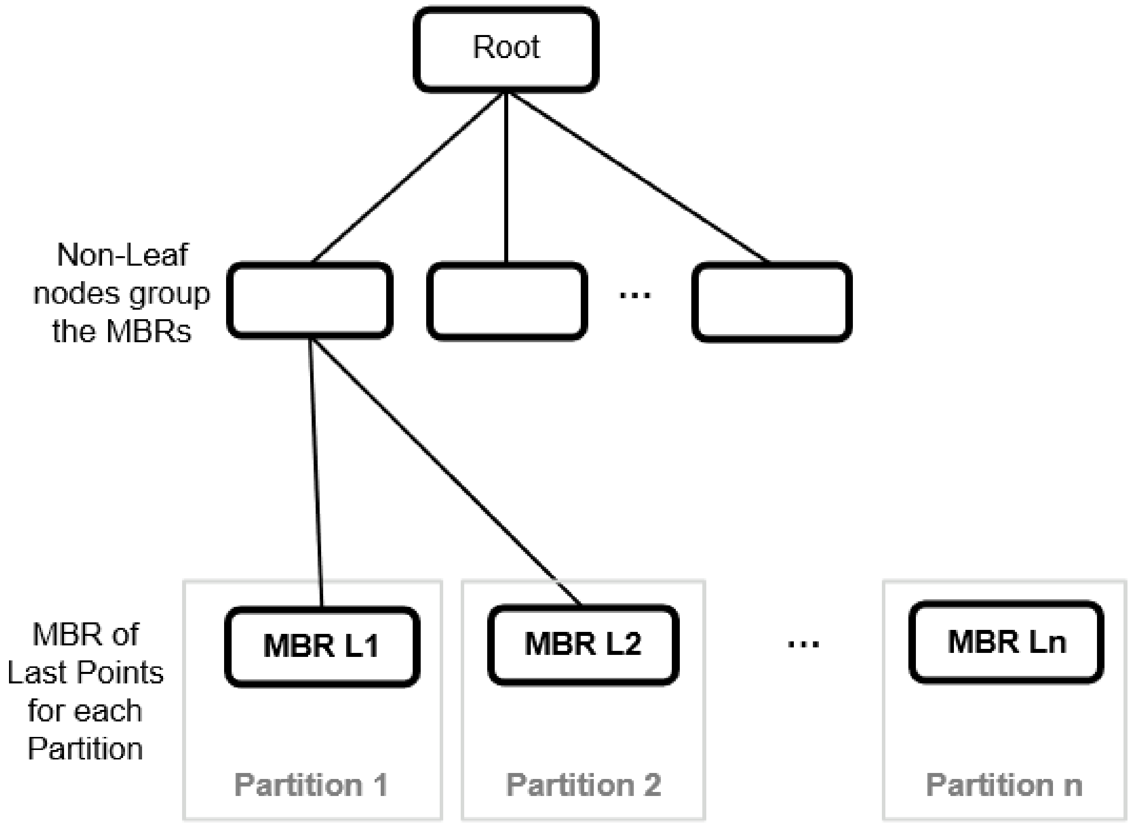

Root
Non-Leaf nodes group the MBRs
...
MBR of Last Points for each Partition
MBR L1
MBR L2
...
MBR Ln
Partition 1
Partition 2
Partition n

Figure 2.6: Global Index R-tree using MBRs of last points

2.2.5 Phase 5: Model Validation for Future Batches

In the last phase of our algorithm, the model associated to each trajectory is validated. For each incoming point p_n belonging to a trajectory t, we measure its distance and angle with existing points $p_1, p_2, p_3, ..., p_n$ belonging to t in an online fashion. The resulting distance and angle are then compared with the average distance and angles of the model for trajectory t and the difference termed as *error* is averaged out among all trajectories that are in the stream.

The phase 5 corresponds to the AvgValidationError function in the pseudo-code of algorithm 2. For points $p_1, p_2, ..., p_n$ in a streaming trajectory t, the *model* contains the trajectory's *heading* vector which comprises of average distance and angle

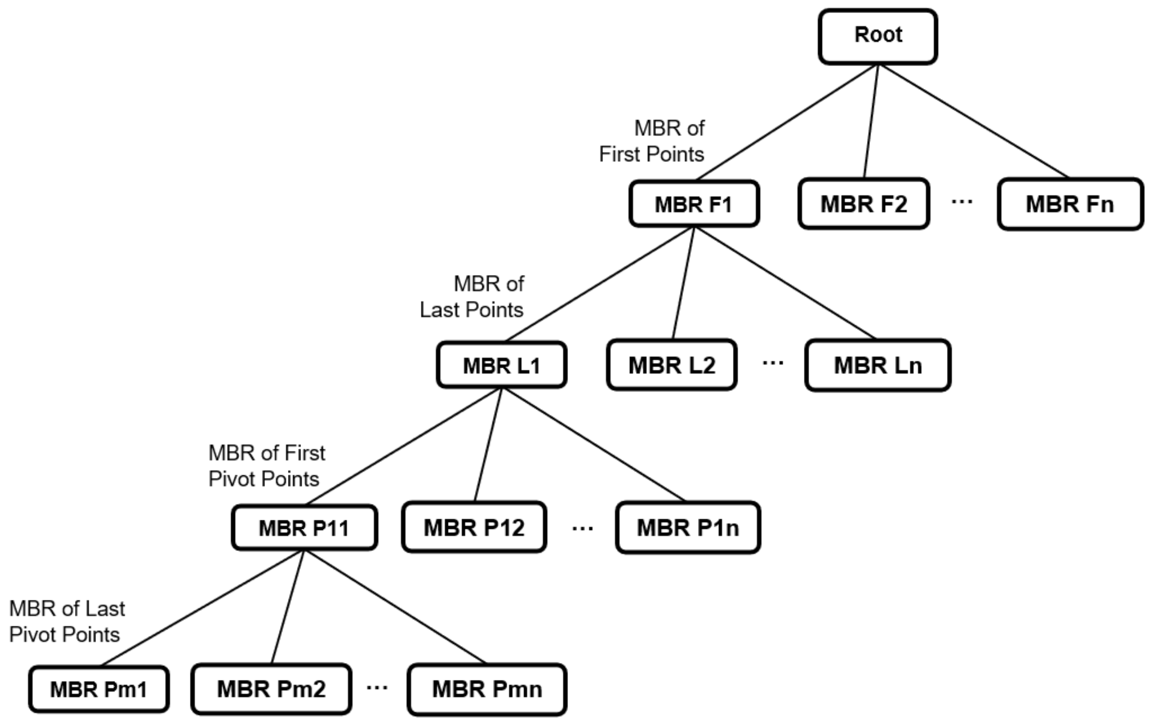


Figure 2.7: Local Index using MBRs of first, last and pivot points

between pair of points in t. For a window of size w, we track a $newHeading$ vector by calculating $avgDistance$ and $avgAngle$ among new points $p_{n+1}, p_{n+1}, ..., p_{n+w}$ in the incoming batches. The validation error for the model is then calculated in terms of Euclidean distance between vectors of $heading$ and $newHeading$. The function returns the average of these errors for all trajectories.

If the error is greater than a user-defined threshold ϵ, the model is updated and the indexes (local and global) are rebuilt from scratch.

3 Experimental Evaluation

This chapter presents the experimental setup, performance analysis, and results. Section 3.1 describes the setup of experiments to evaluate our algorithm, PIMMLI. We present our performance analysis in section 3.2, which provides details of the datasets, hardware systems, competing techniques, evaluation metrics and algorithm parameters used for our experiments. Section 3.3 provides results from the experiments performed on our algorithm. These include experiments observing the impact of important factors presented in 3.2 on PIMMLI.

3.1 Experimental Setup

PIMMLI is built on Apache Spark Streaming [1], leveraging this streaming framework for distributed in-memory computation. This allows real-time data processing through various data sources such as Apache Kafka, Kinesis, Flume, HDFS, etc. We use file-system-based streaming, where a certain data source directory can be monitored by the streaming API.

A *Python* script is used to simulate file streaming. This script generates new files that represent the newest batch in the stream to the algorithm. The copies of files in a defined format contain new points for each trajectory. Each file copy contains trajectories separated by lines, each trajectory contains points separated by a semicolon as a delimiter, and each point contains 2 values corresponding to its

[1]https://spark.apache.org/streaming/

latitude and longitude separated by a comma.

The version of libraries, frameworks and languages that we used to build PIMMLI and perform experiments are *Scala 2.11, Python 3.8, Open JDK 1.8, Spark 2.4.4, SBT 1.3.4*, and *Maven 3.6.2*.

3.2 Performance Analysis

This section will present the different configurations on which the performance of the competing algorithms were tested. These include datasets, hardware, evaluation metrics, and algorithm parameters.

3.2.1 Datasets

We have used three real-life datasets: GeoLife, Porto, and Beijing. These datasets are comprised of different activities and scale ranging from largest (GeoLife) to smallest (Beijing). This subsection will provide details of each dataset with a visualization of sampled trajectories.

GeoLife

Microsoft Research Asia collected data for outdoor activities of 178 users and published it as GeoLife dataset [104]. It spans over approximately 0.8 million miles spatially and 48k hours temporally, collected in a period of roughly 4 years. The dataset originally contains 17k trajectories. After splitting large trajectories into several smaller trajectories, we also filter out any original trajectories which are very small i.e. those with points count less than the amount we are streaming for the duration of the entire experiment. After such pre-processing, we extract 50k trajectories with 12.5 million points to be used in our experiments. The trajectories are spread out across Beijing, China as shown in figure 3.1. This dataset, as opposed to the other two, represents several outdoor activities, such as cycling, walking, hiking, sightseeing and shopping. Due to the diverse range of activities covered and to it being the largest dataset among the ones used in this thesis, we have used this dataset for our default experiment configuration.

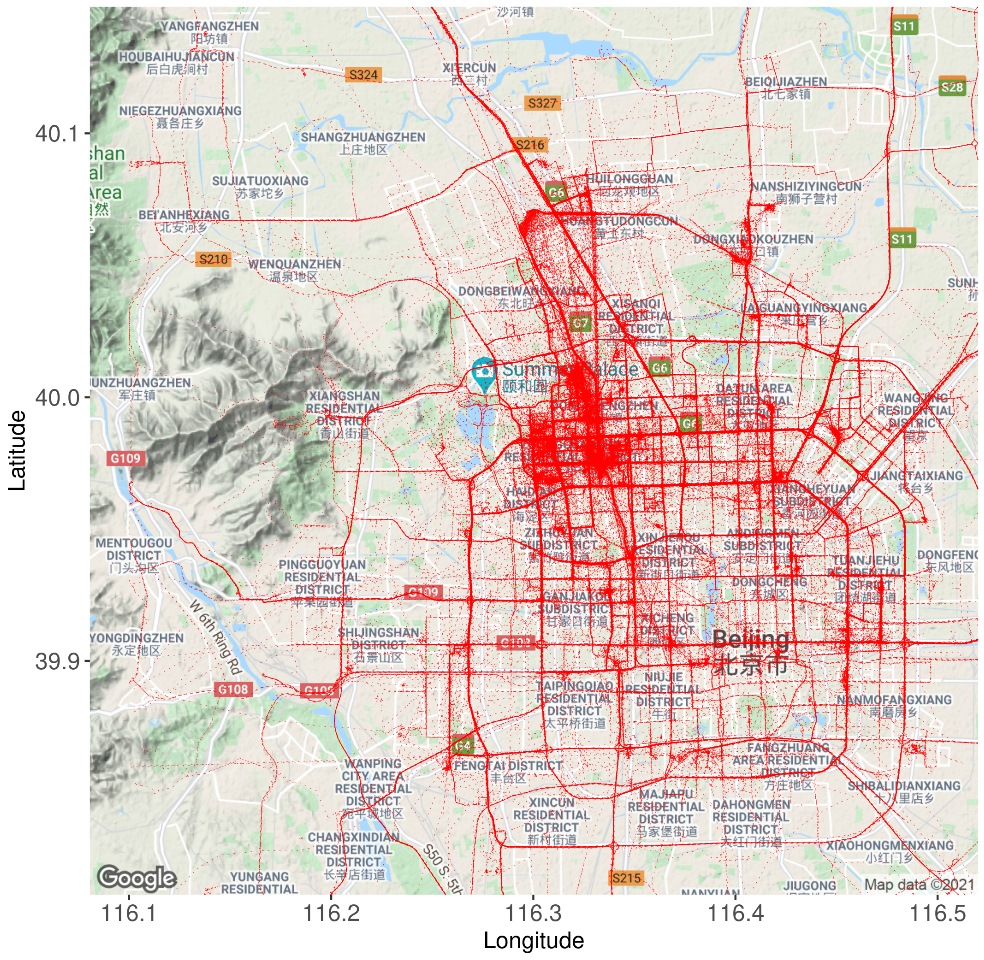


Figure 3.1: Sample of the GeoLife dataset trajectories mapped over Beijing, China

Porto

The Taxi Service Trajectories dataset contains 442 Taxis running in the city of Porto, Portugal [60]. It was published by the 2015 data mining challenge of ECML PKDD (European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases) [2]. The duration of data collection is approximately a year and is well spread out in the city, as shown in figure 3.2. We performed similar pre-processing to this dataset as we did for GeoLife: we split larger trajectories into several smaller ones and filtered out any trajectories with fewer than batch count number of points. After such pre-processing, we are left with 9,387 trajectories containing 2.35 million points in total.

Beijing

Our third dataset is the *Beijing* dataset, also known as the T-Drive dataset [96]. This dataset contains the trajectories collected over a period of 1 week of 10,357 taxis. We are using a randomly sampled version of this dataset, which one of our competing techniques, DITA [74] used in their experiments and shared along with their repository. The scaled down version after pre-processing ends up having 43 trajectories containing 2,150 points in total, representing the smallest dataset for our experiments. Figure 3.3 shows the Beijing taxi dataset mapped over the city of Beijing. To ensure that most streets are visible in the figure, we have mapped 30% of the dataset randomly sampled with the uniform distribution.

[2]http://www.geolink.pt/ecmlpkdd2015-challenge/

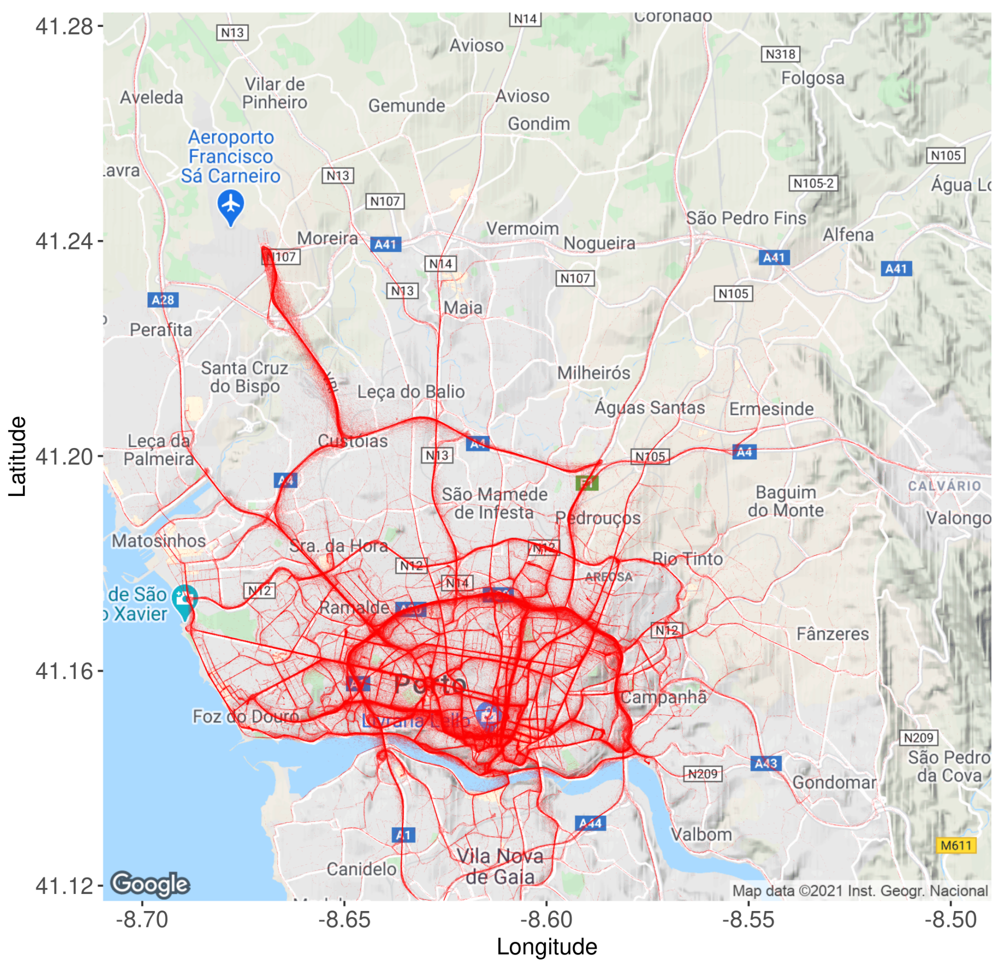


Figure 3.2: Sample of the Porto taxi dataset trajectories mapped over Porto, Portugal

3.2.2 Hardware

We used two systems provided by the Computer Science Department of the University of Minnesota Duluth, namely Janus and Ukko. Both systems used Ubuntu 18.04 at the time of experiments. The hardware configuration of each system is listed below:

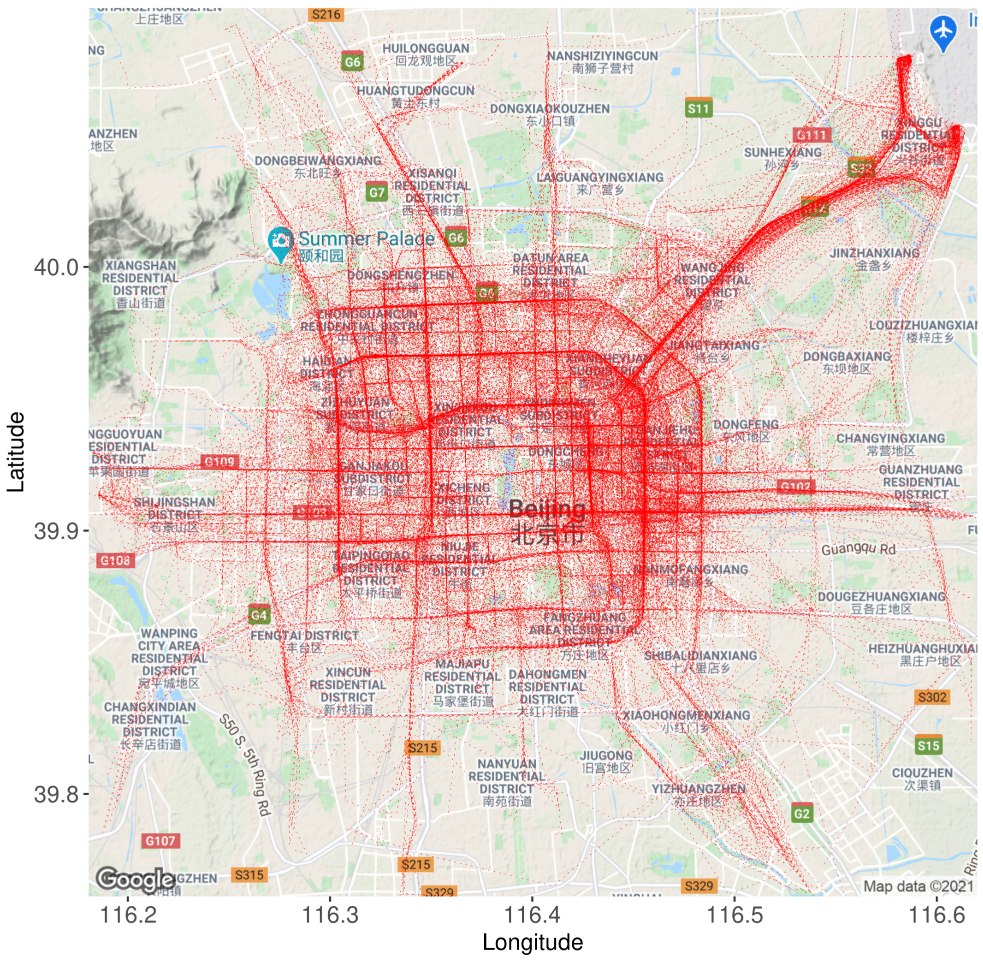


Figure 3.3: Sample of the Beijing taxi dataset trajectories mapped over Beijing, China

Janus

Janus runs on a 3.2 GHz Intel Xeon Gold 6134 CPU. It has 8 cores per socket with 2 sockets and 2 threads/core. Therefore our experiments involve spawning worker nodes across 32 different threads as provided by the system. The system contains 512 GB of RAM. It also has an Nvidia Tesla V100 GPU installed with 16 GB of RAM.

Ukko

Ukko has 2.6 GHz Intel Xeon E5-2690 CPU and an Nvidia Tesla P100 GPU. It contains 14x2 cores and 56 threads. There is a 512 GB Memory installed and 12 GB of Graphic RAM.

3.2.3 Competing Techniques

Our algorithm is compared against two techniques, naive and DITA [74]. Our setup includes in-memory computation for all techniques for a fair comparison.

Naive

Our naive technique is a non-indexed approach. As the batches are streamed, they are stored directly without any indexes, as shown in algorithm 4. As such, any query issued on available trajectories exhaustively evaluates all points in the database and ongoing stream. Figure 3.4 demonstrates a median total execution time of 10 range queries issued at each batch which introduces equal number of points. The effect can be cumulatively observed in figure 3.5.

Algorithm 4 Naive Approach

```
Require: db: Previously Streamed Trajectories, b: Newest Batch in Stream

1: function ProcessBatch(db, b)
2:     $db \leftarrow db \bigcup b$
3:     Queries search in db exhaustively
4: end function
```

DITA

Distributed In-Memory Trajectory Analytics, also known as DITA [74], introduces spatial indexing of trajectories in batch-oriented systems. As this technique is batch-

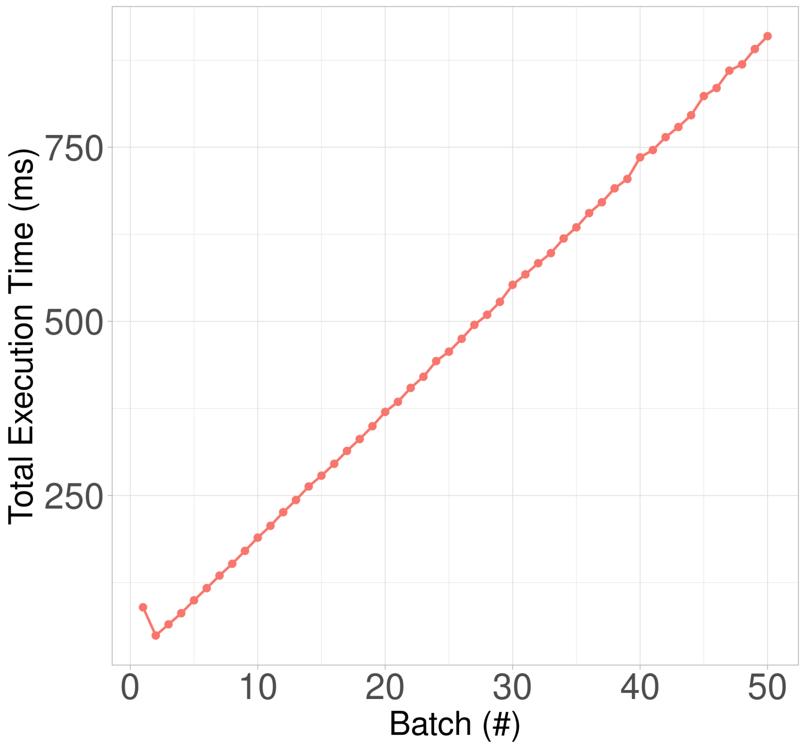


Figure 3.4: Total execution time using the naive approach

oriented, it uses the entire data at once for processing. The dataset is partitioned using *STR partitioning* [51] and then indexed in a distributed environment. To set a fair baseline where each competing approach can work on streams, we integrated Spark Streaming into DITA and indexed all available trajectories at every incoming batch as shown in pseudo-code 5. The points in each incoming batch are appended to their respective trajectories and the combined database of trajectories is processed by DITA's existing batch-oriented algorithm at line 3 of the pseudo-code 5.

Algorithm 5 DITA extended for streaming

Require: db: Previously Streamed Trajectories, b: Newest Batch in Stream

```
1: function ProcessBatch($db, b$)
2:     $db \leftarrow db \bigcup b$
3:     DITA($db$)
4: end function
```

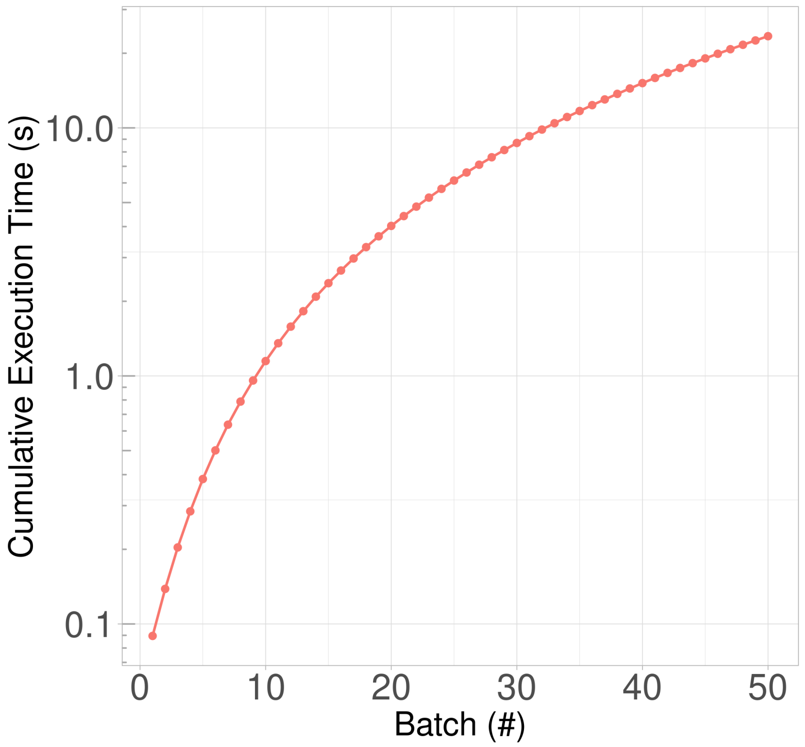


Figure 3.5: Cumulative execution time using the naive approach

Figure 3.6 shows a median total execution time of indexing and 10 range queries as we add new batches. The spikes in performance indicate the allocation of additional partitions when distant points of trajectories are streamed. Cumulatively, the execution time trend can be observed in the log-plot of figure 3.7.

3.2.4 Evaluation Metrics

We evaluate PIMMLI and DITA in terms of indexing time and combined execution time of 10 range queries i.e. We execute 10 range queries every time we index an incoming batch of new points. The total execution time contains indexing time (time from when a batch arrives to the time it is completely indexed) and query time (time from issuance of a query to the time results were received for all executed queries). An experiment involves streaming 50 batches, and in each batch, we perform indexing

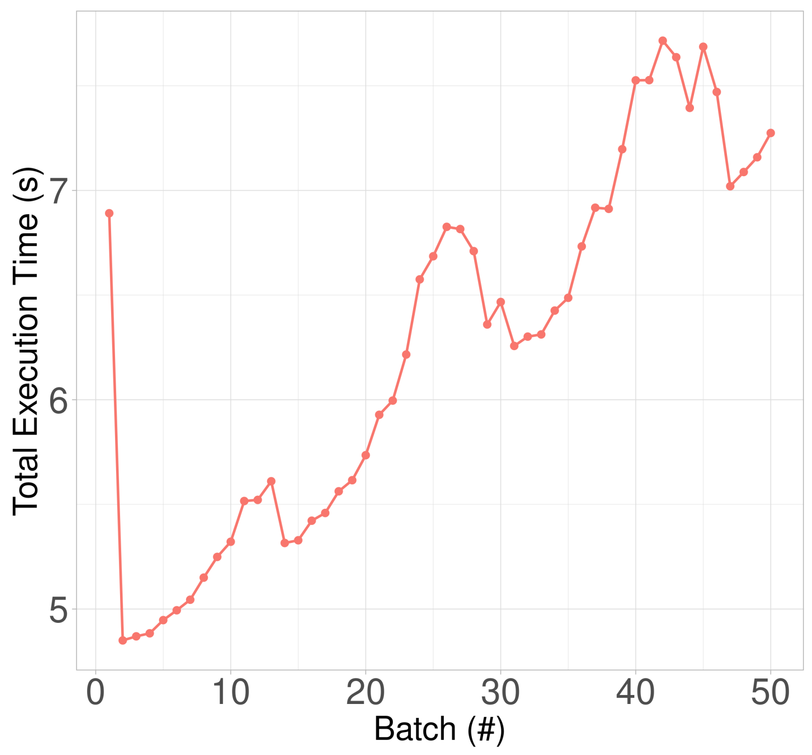


Figure 3.6: Total execution time using DITA in streaming

on new points and execute 10 queries. The naive approach, since it is a non-indexed method, is evaluated only in terms of executing 10 queries. All experiments are performed 5 times and a median of execution times is taken for each of these metrics at each batch. Cumulative execution time at any given batch b is the sum of all total execution times in previous batches $1, 2, ..., b-1$ and the current batch b.

3.2.5 Parameters

Table 3.1 lists the range of values at which PIMMLI is tested for each parameter while default values are kept for other parameters. Here we provide more details for each parameter.

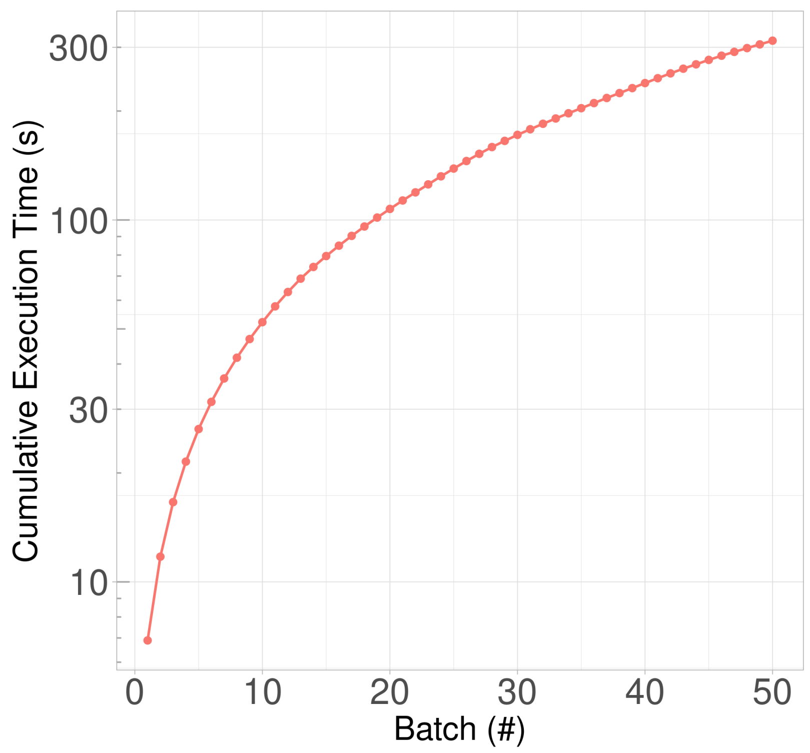


Figure 3.7: Cumulative execution time using DITA in streaming

Parameter Name	**Range of Values**	**Default Value**
w **(Window Size)**	1, 2, 3, 4, 5	5
f **(Future Point Count)**	10, 20, 30, 40, 50	20
ϵ **(Validation Error Threshold)**	0.05, 0.10, 0.15, 0.20, 0.25	0.10
t **(Training Interval)**	10, 20, 30, 40, 50, 100	100

Table 3.1: Experiment parameters

Window Size (w)

The window size (w) represents the number of consecutive batches that will be used for model training and validation. As such, a window size of 5 (default value) implies that 5 batches need to be initially streamed before we use them for any model training or index building. For any incoming batches, the window of size w contains the most recent w batches of data-stream. We chose smaller window sizes, ranging from 1 to 5 in intervals of 1, as streaming applications in practice are expected to have almost real-time processing of incoming batches [85], and as such a large window size requires more batches to be received before processing.

Future Point Count (f)

Once the model is trained, it is then used to predict future points for each trajectory, as explained in sub-section 2.2.3. f represents the number of such points predicted. For its default value, 20, the trajectories used for indexing contain their existing points appended with 20 additional points. We chose values in the range of 10–50, in intervals of 10, as too small a value (<10) will cause a smaller investment in future and hence a repetitive re-training of the models, and a larger number may not be a good representative of a trajectory in the long term [69].

Validation Error Threshold (ϵ)

The model is validated for subsequent windows of batches such that if the error is greater than ϵ, we retrain the model and update the indexes. This allows us to account for certain unpredictable patterns in trajectories and update the internal representation whenever necessary. A range of Euclidean distance values between 0.05–0.25 in intervals of 0.05 were chosen for ϵ. It represents the average distance between points in all trajectories of our default dataset for experiments (GeoLife [104]).

Training Interval (t)

Depending on the nature of trajectory data and the availability schedule of computational resources, it may be desired to update the model every certain number of batches. t specifies such interval at which the model will be re-trained even if the error has not reached the ϵ threshold. We chose values 10, 20, 30, 40, 50, and 100 as our default streaming batch count for experiments is 50 and it may be interesting to observe model re-training happening at different points during that stream. The 100 value of t is aimed to show the execution time patterns when re-training is not pre-planned for current duration of the stream.

3.3 Experimental Results

This section presents results from the experiments comparing our algorithm, PIMMLI, against DITA and the Naive approach. These include experiments observing the impact of different datasets, hardware systems, trajectories count, and algorithm parameters such as Window Size w, Future Point Count f, Validation Error Threshold ϵ and Training Interval t.

3.3.1 Impact of Datasets

Different datasets represent different characteristics of data. The GeoLife dataset [104] represents data of various activities such as walking, cycling, biking, etc. collected in a very long duration (approximately 4 years) whereas the Porto dataset [60] contains trajectory data only from taxis collected in a very short duration (approximately 1 week). We also have variation in cardinality, where the Beijing dataset [96] with 2,150 considered points is the smallest and the GeoLife dataset with 12.5 million points is the largest dataset used in our experiments.

Figure 3.8 shows the performance over our smallest dataset. This figure shows that the Naive approach seems to perform better than DITA. In contrast, the larger datasets show a different trend in figures 3.9 and 3.10. As shown in the figures, PIMMLI consistently performs better than both of the competing approaches at all datasets.

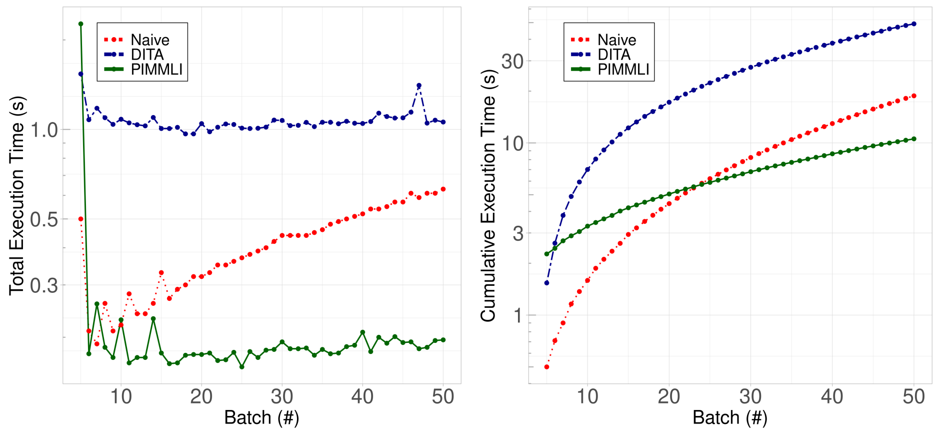


Figure 3.8: Performance over the Beijing taxi dataset

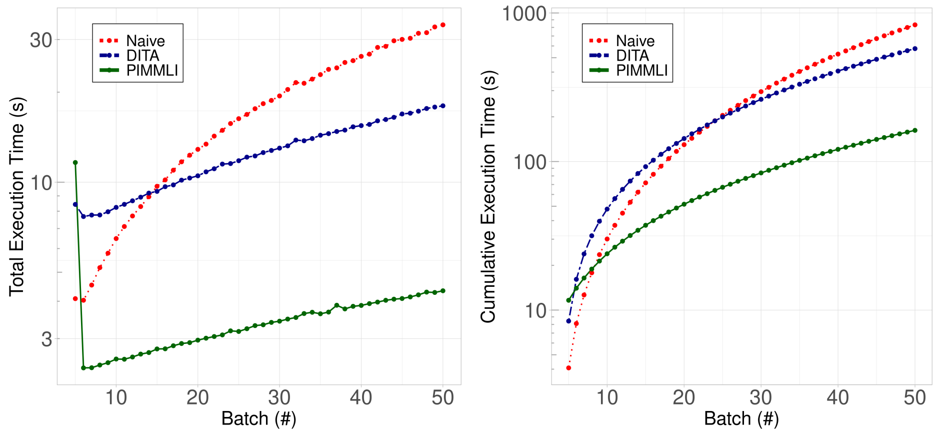


Figure 3.9: Performance over the Porto taxi dataset

On the Beijing dataset, PIMMLI performs on-average 5X better than DITA as observed by cumulative performance in figure 3.8. On the other hand, for larger dataset such as Porto, PIMMLI cumulatively performs 3.3X better than DITA as demonstrated by figure 3.9. Similarly, there is a 2.9X factor at the GeoLife dataset as shown by figure 3.10. An explanation for the difference of these performance factors is that simpler models like the one employed by PIMMLI are more robust in predictions for a smaller dataset, although they still outperform the non-predictive approaches

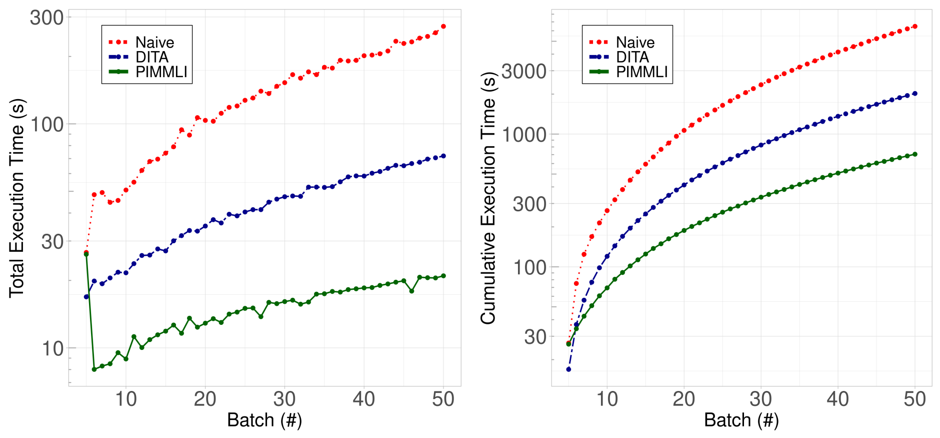


Figure 3.10: Performance over the GeoLife dataset

such as the DITA and Naive algorithms.

3.3.2 Impact of Trajectory Counts

The trajectory count is the number of trajectories being streamed. We varied the count of trajectories while keeping the batch size fixed, i.e. the total number of points streamed in the experiment was kept constant. The trajectory counts are varied among 10k, 20k, 30k, and 40k, while the complete batch size stays as 120k points in each case. Figures 3.11 and 3.12 show that PIMMLI is spatial characteristics-agnostic i.e. it shows similar performance in lesser trajectories yet increased number of points which can be relatively closer in a trajectory, as with lesser points yet increased number of trajectories that represent a completely differently route and potentially distant points.

In the first experiment, shown by top-left figure with label 10k, we added 12 points per trajectory and hence the new points added per batch were 120k. As we increase the number of trajectories in later experiments, we decreased the number of points per trajectory. For 20k trajectories, we added 6 points per trajectory. Similarly, 4 and 3 points per trajectory for 30k and 40k trajectories respectively, keeping the

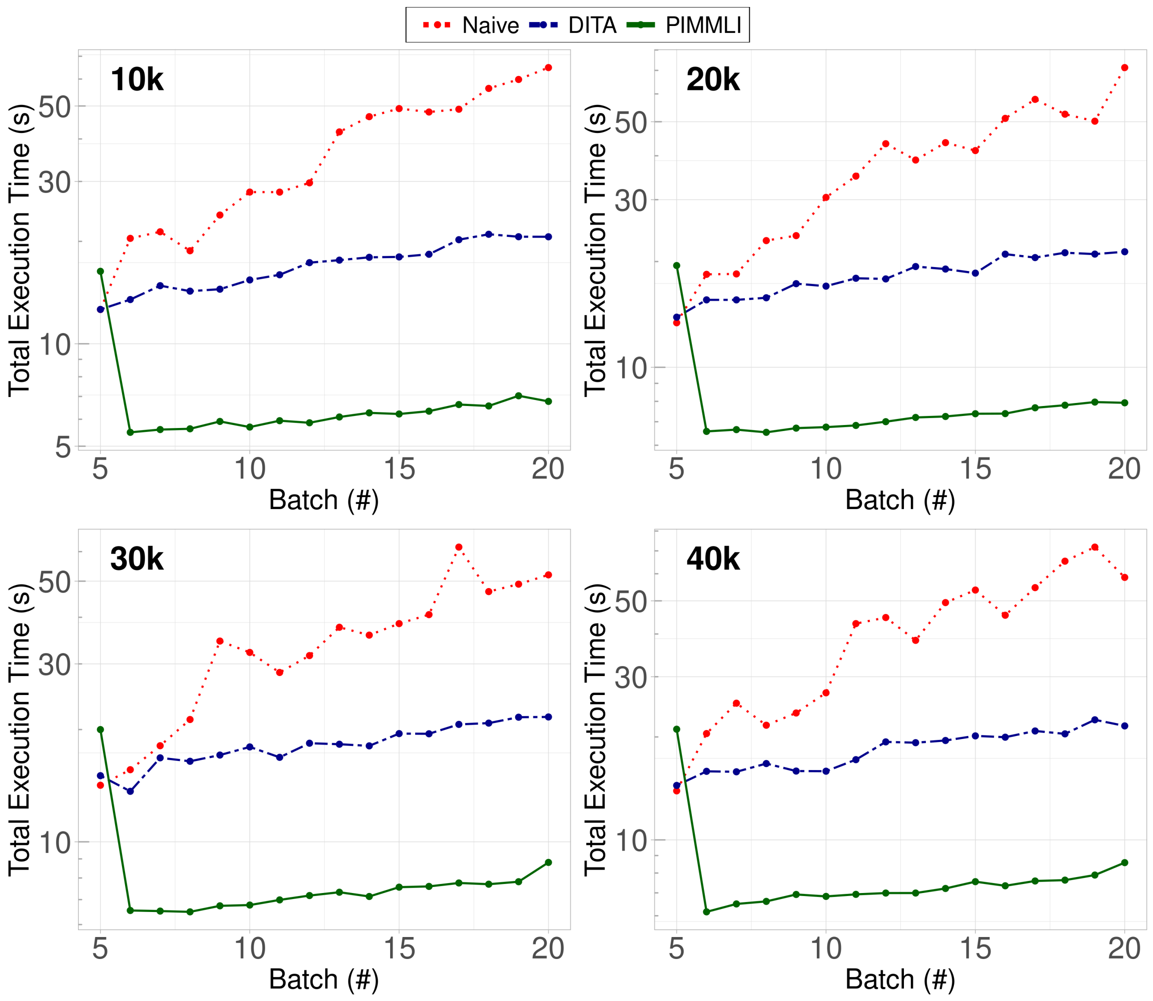


Figure 3.11: Total impact of trajectories count

batch size constant at 120k points per batch. All 4 sub-figures in figures 3.11 and 3.12 show similar performance of PIMMLI whereas some performance variations can be observed in the DITA and Naive algorithms.

3.3.3 Impact of Hardware

We observe that different systems showed similar performance trends, as shown in figures 3.13 and 3.14. However, there is a multiplicative factor in difference of

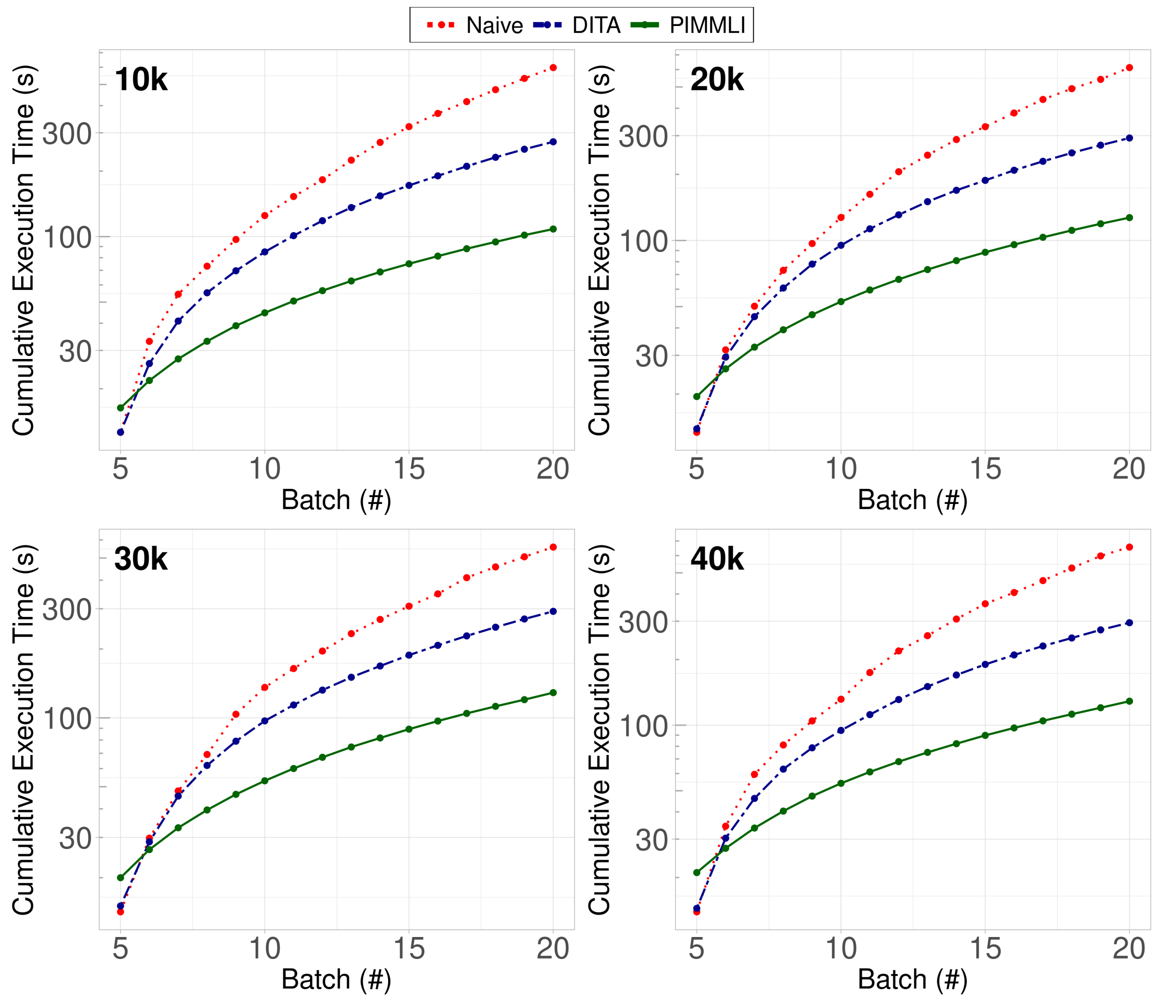


Figure 3.12: Cumulative impact of trajectories count

performance based on available number of worker nodes and memory within each of them. This factor is more noticeable in the cumulative execution time results where PIMMLI performs approximately 3X faster at Janus than Ukko.

The difference could be because of the memory available for each worker node at different systems. With more threads in Ukko than in Janus, yet both have same the amount of RAM available, there are more nodes at Ukko but less memory available for each node. Janus, on the other hand, has more RAM available for each worker

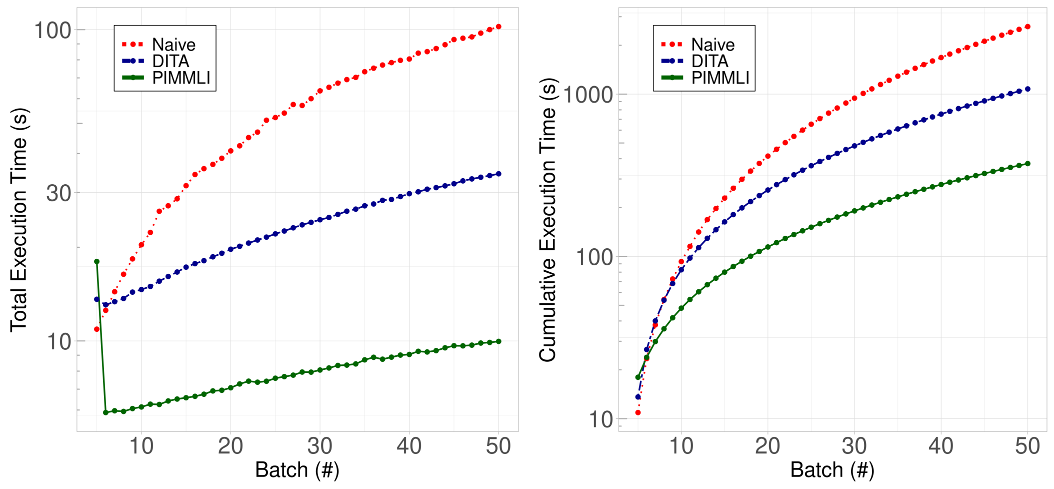


Figure 3.13: Performance at Janus

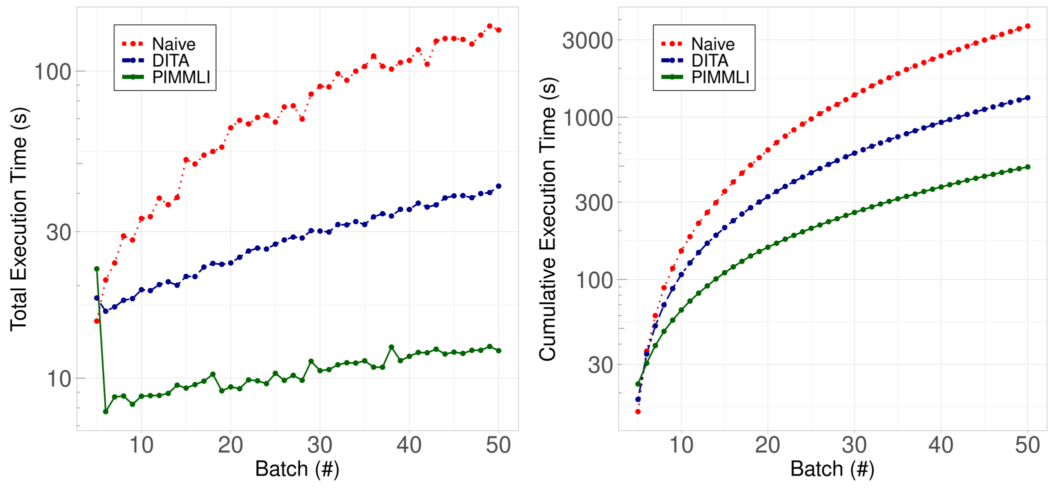


Figure 3.14: Performance at Ukko

node and hence allows more in-memory computations.

3.3.4 Impact of Window Size

The Window size (w) is specified in terms of number of batches to be present in a window for model training and validation. The experiment is performed using a 1–5 range of w in intervals of 1, while using default values for the other parameters listed in table 3.1. As demonstrated by figures 3.15 and 3.16, we observe no significant effect on these sizes as they are relatively smaller than the number of points we predict for future.

As we set $w = 1$, 1 batch at a time is processed for model training/validation and indexing if needed. As observed from figure 3.15, PIMMLI has a total execution time of approximately 5 seconds on average at each batch. As we increase w, which means that more batches are streamed before a window processes the new batches, we do not observe any performance change at these values. This can also be observed by cumulative results in figure 3.16 as PIMMLI takes around 300 seconds at all values of w by the end of stream at 50^{th} batch.

3.3.5 Impact of Future Point Count

For each trajectory, PIMMLI predicts its future points so that the trajectory can be indexed efficiently for streams. For this experiment, we increase the future point count in samples of 10, 20, 30, 40, and 50. As we increase future point count, it costs more in terms of recursive prediction for each trajectory which can be observed by offset of execution times in figures 3.17 and 3.18.

For $f = 10$, the total execution time for each batch on average stays around 5 seconds, as shown in figure 3.17. As we increase the value of f, the execution time increases as well. For $f = 20$, it jumps to around 6–7 seconds. Similarly, it takes 8–9, 9–10, and 10+ seconds on 30, 40, and 50 values of f respectively. The competing approaches stay unaffected as they do not use this parameter. The impact of future point count on PIMMLI's performance is more noticeable by cumulative results in figure 3.18. It can be observed that PIMMLI takes around 250 seconds at $f = 10$ whereas it is 500+ seconds at $f = 50$, at the end of 50 batches in stream.

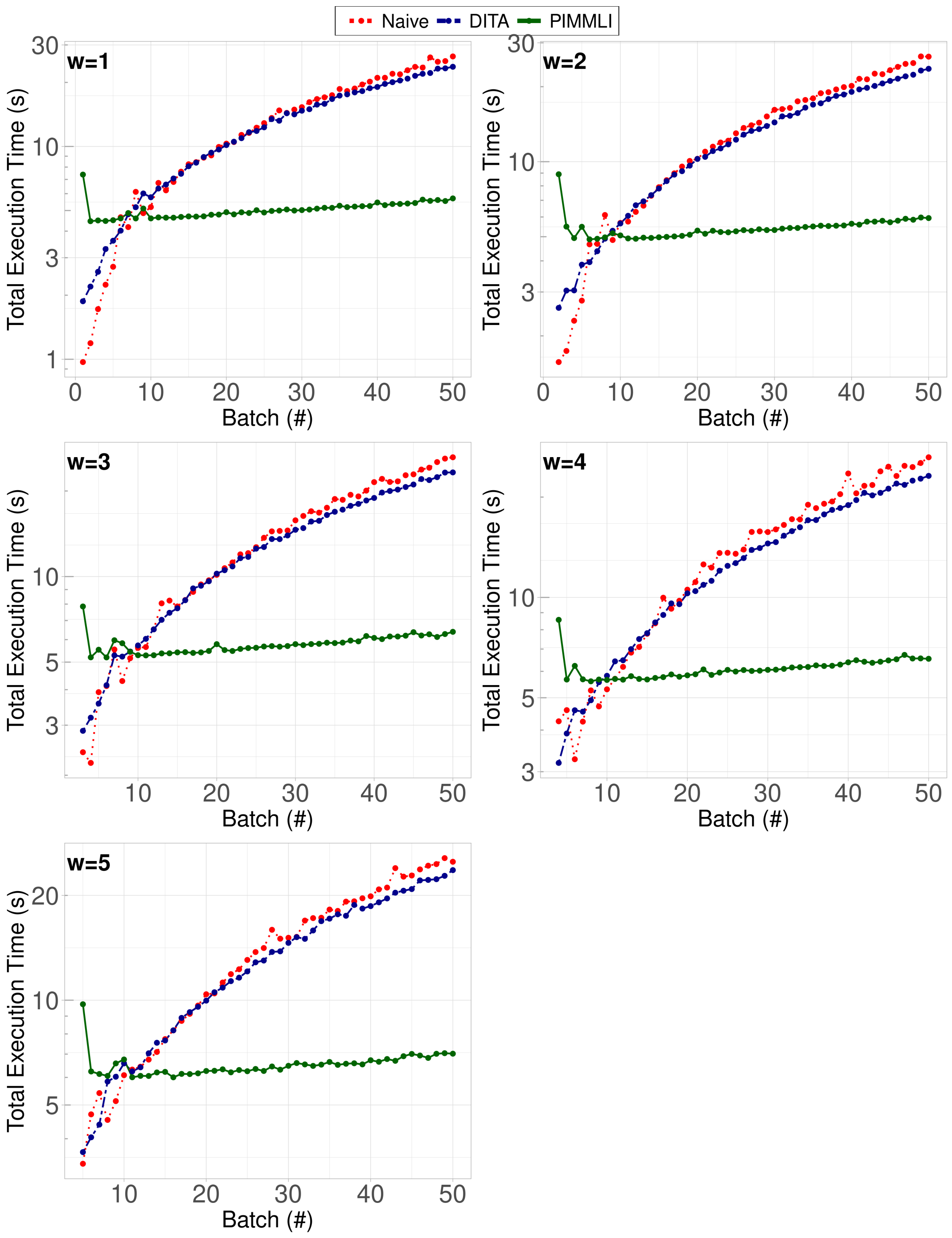


Figure 3.15: Total impact of window size (w)

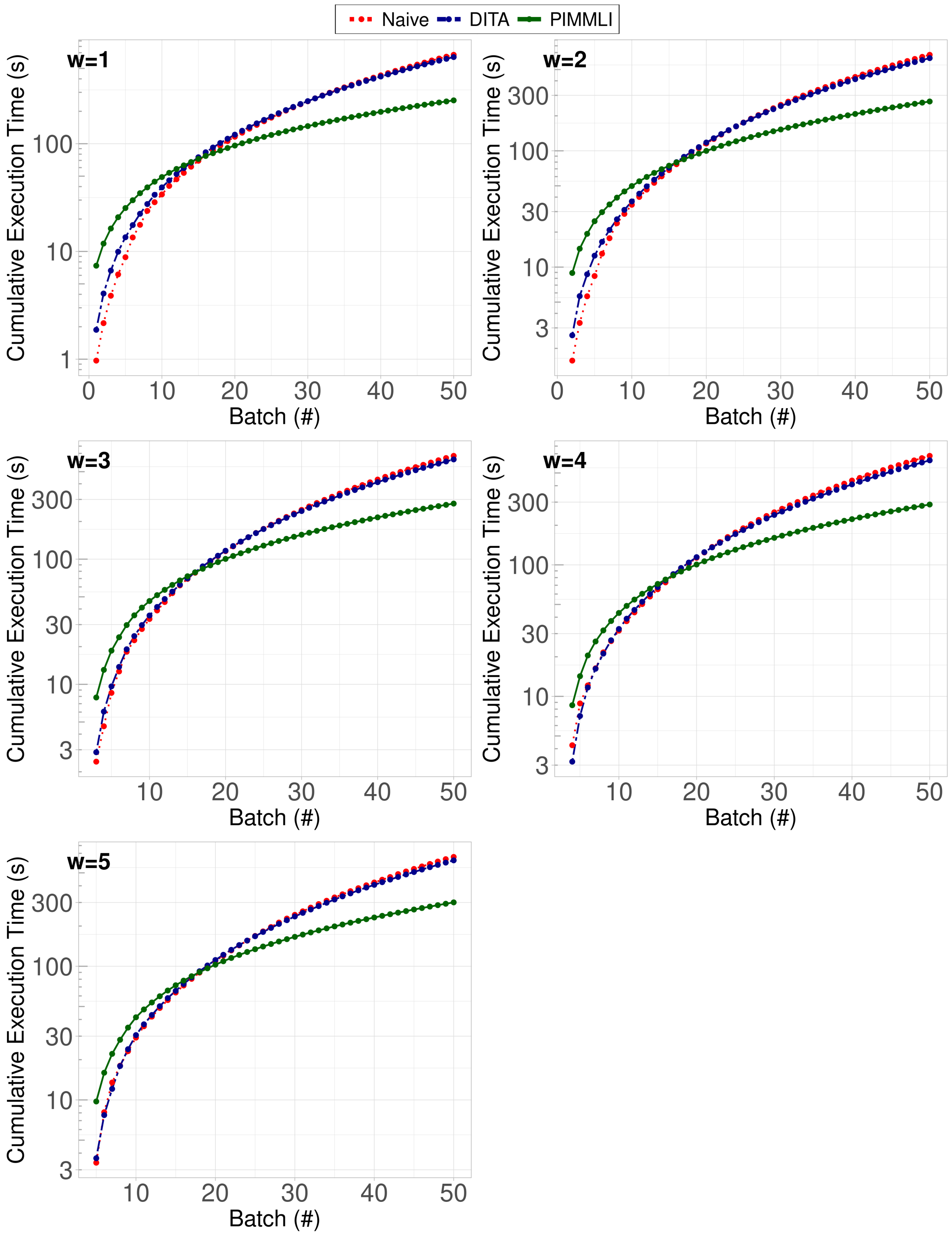


Figure 3.16: Cumulative impact of window size (w)

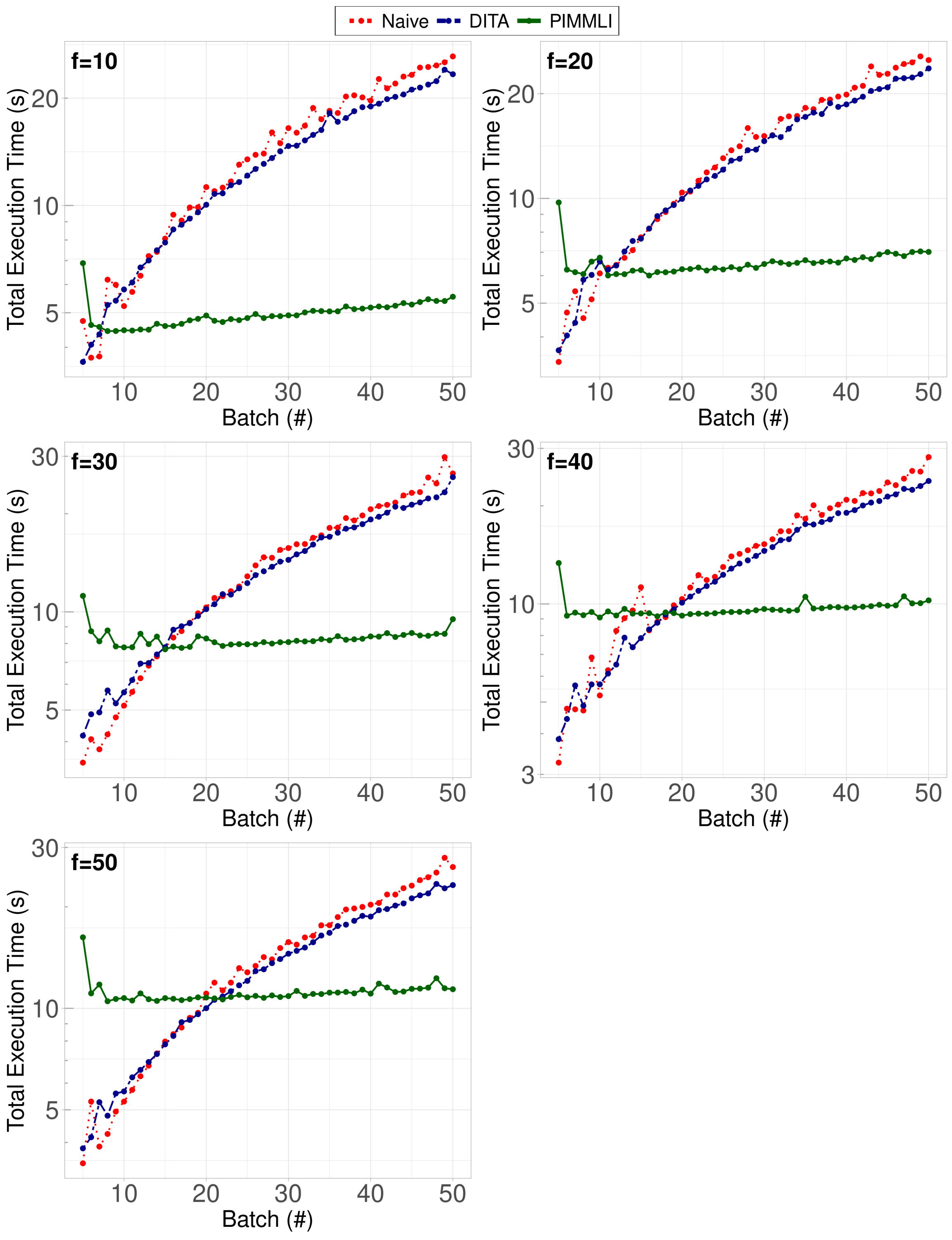


Figure 3.17: Total impact of future point count (f)

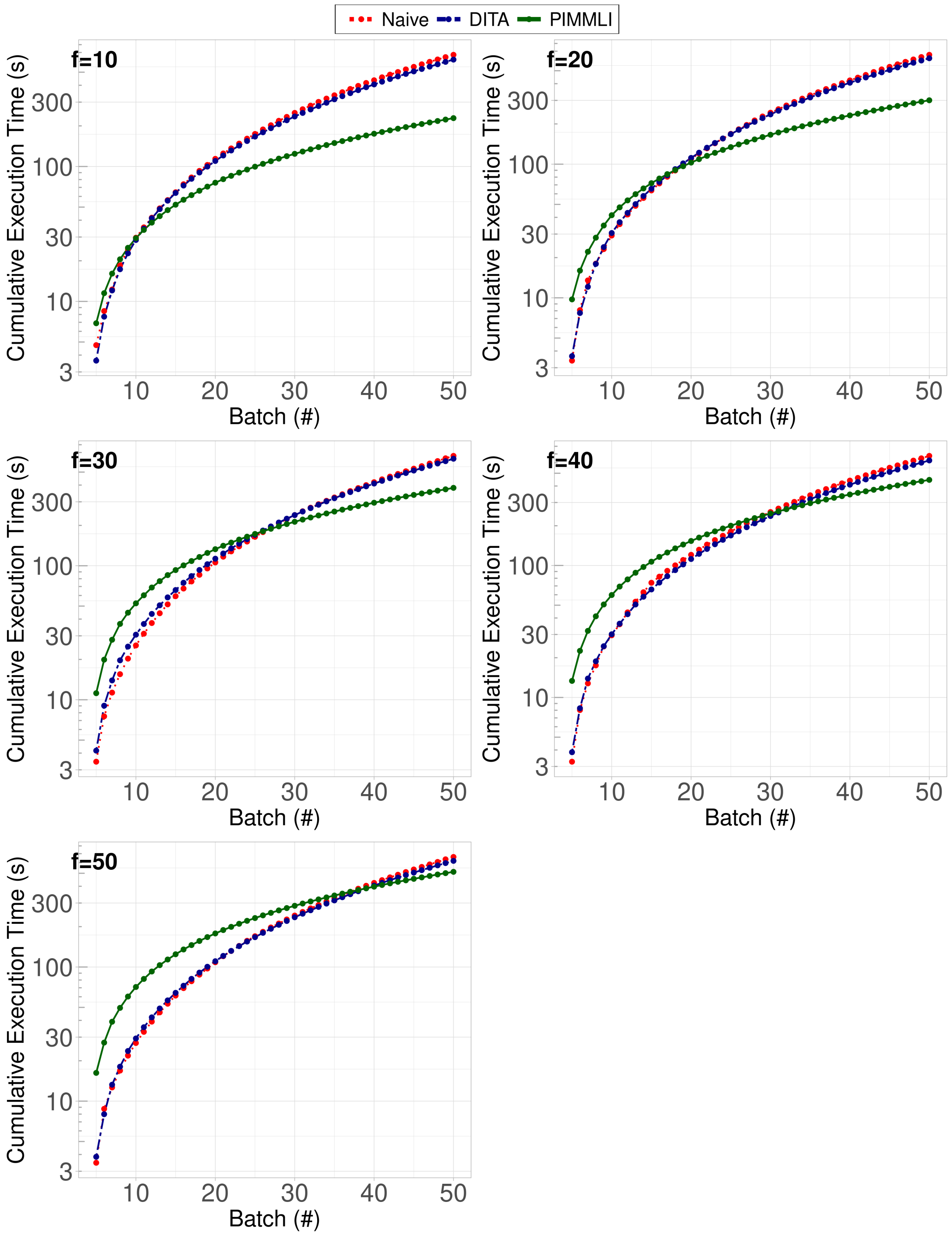


Figure 3.18: Cumulative impact of future point count (f)

3.3.6 Impact of Validation Error Threshold

The Validation Error Threshold (ϵ) sets a target average error for incoming window's validation. If the model of a trajectory has an average error less than (ϵ), the model is retrained for a trajectory. This ensures that if there are certain trajectories in a stream for which its model's internal representation does not predict the future of that trajectory accurately, then such models should be frequently updated taking into account more data points in general.

We try values in the range of 0.05–0.25 with an interval of 0.05. This number in terms of Euclidean distance roughly translates to 2.3 miles when a geo-location dataset is used. In the case of the GeoLife dataset [104], it represents an average distance between consecutive points of trajectories, thus specifying the error threshold in multiples of such average distance simplifies the choice of ϵ value as it represents by how many points on average a trajectory can be off in a prediction. By experiment, we observe small peaks at smaller thresholds, as shown in 3.19, especially for the smallest value of ϵ i.e. 0.05. However, this effect appears negligible for all values of ϵ when observing the cumulative execution time impact as shown in figure 3.20.

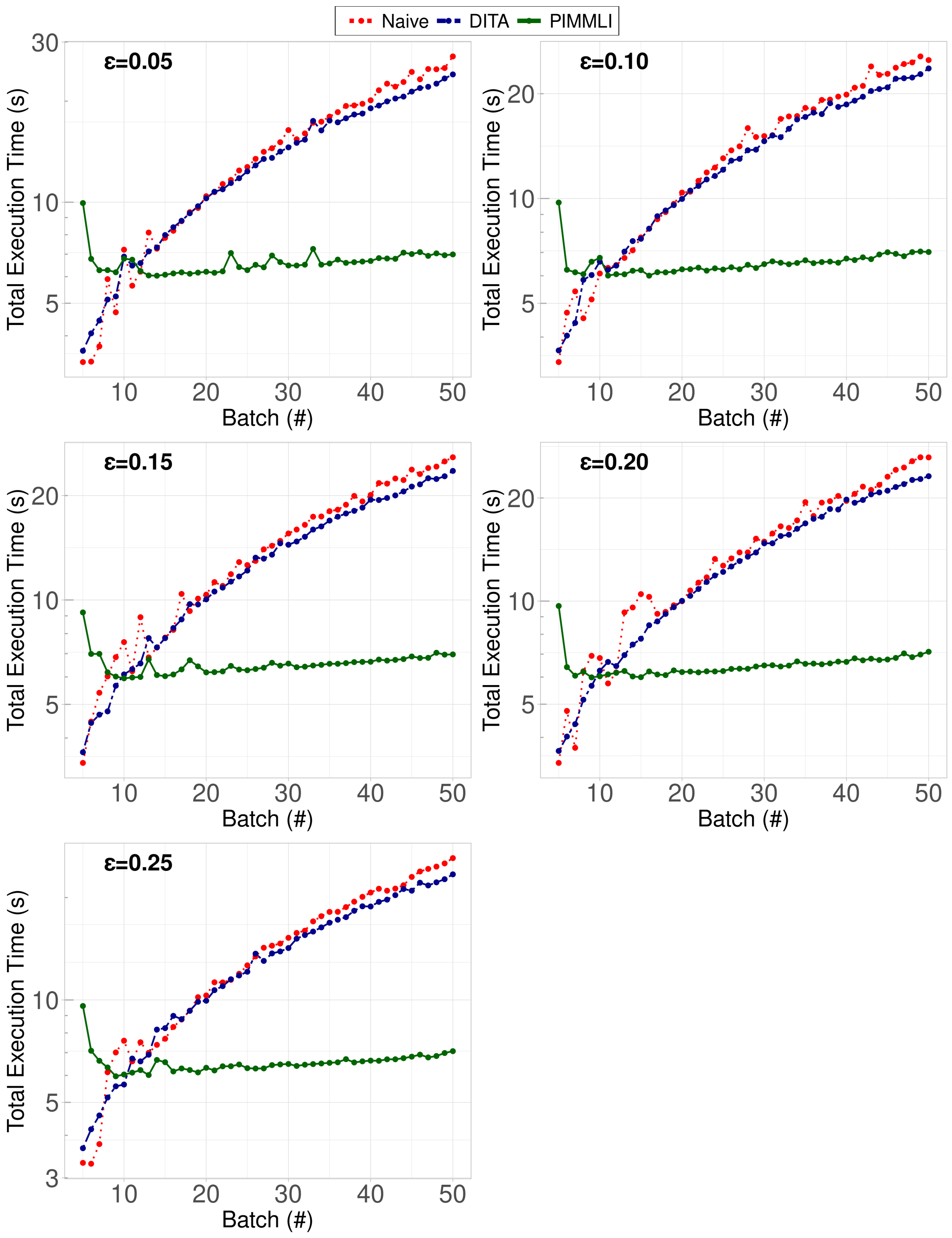


Figure 3.19: Total impact of validation error threshold (ϵ)

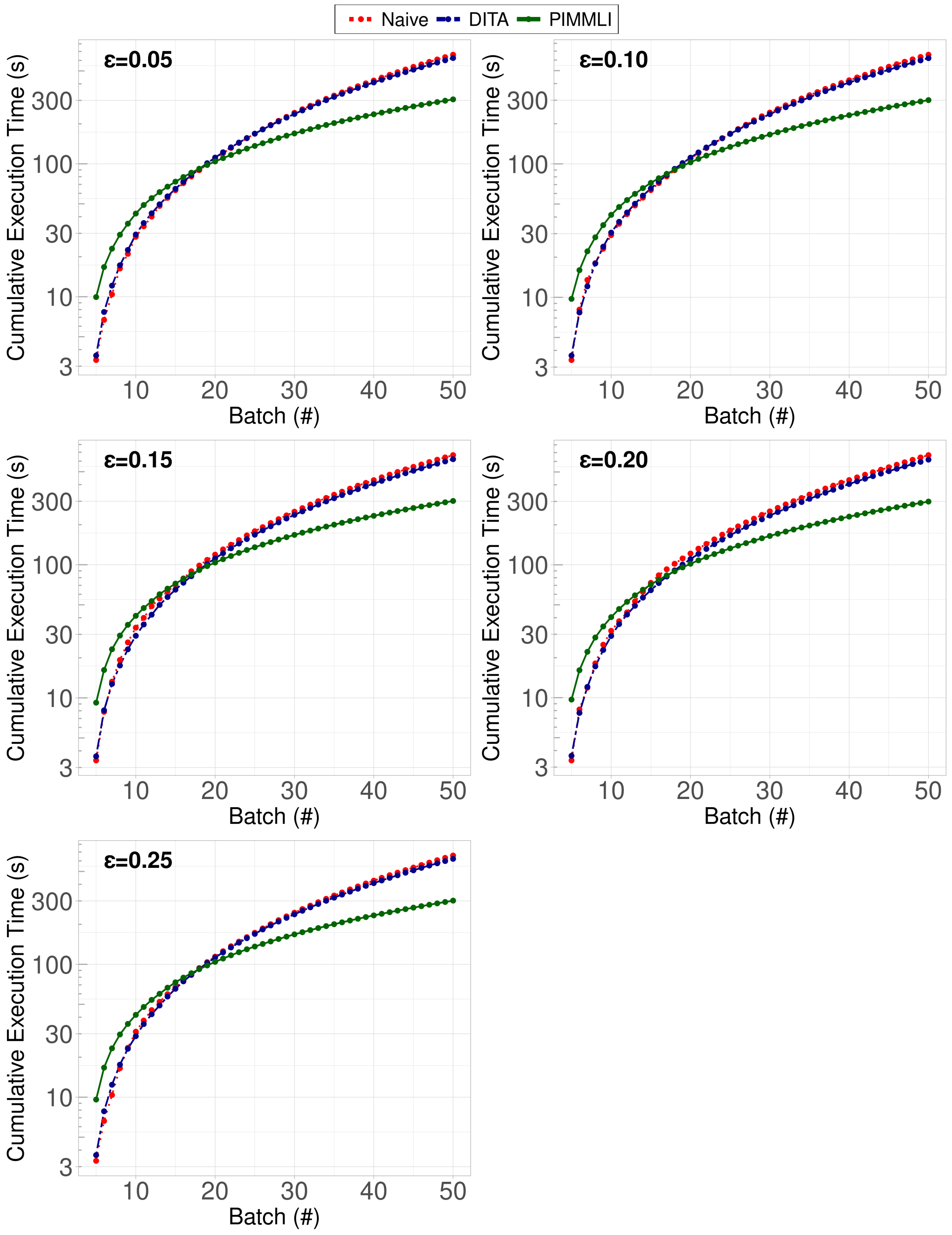


Figure 3.20: Cumulative impact of validation error threshold (ϵ)

3.3.7 Impact of Training Interval

The training Interval (t) makes sure that each trajectory model is re-trained every certain number of batches. As such, t is defined in terms of number of the batches. We have taken values of 10, 20, 30, 40, 50, and 100. The range of 10–50 is chosen to observe the effect at different points within the range of our streaming batch limit (50). The value $t = 100$ demonstrates the effect of the complete training occurring only once in complete stream cycle. Figure 3.21 shows noticeable peaks that occur due to model re-training of all trajectories at every t interval.

It is to note that in all cases, the model is trained only when the first complete batch has been streamed and hence it possesses that peak at the beginning. At $t = 10$, the models are to be re-trained and indexes are rebuilt every 10 batches of streams, therefore, we observe a second peak at the 10^{th} batch, a third at the 20^{th} batch, and so on until we observe the last peak at the 50^{th} batch. For $t = 20$, the second and third peaks occur at the 20^{th} and 40^{th} batches respectively. Similarly, second peaks are observed at the 30^{th}, 40^{th}, and 50^{th} batches for 30, 40, and 50 values of t. We do not see any peaks other than the first one at 100 value of t for the reason that it is planned to occur in intervals of every 100 batches and we streamed only 50 batches in our experiments.

These experiments show that the model, if trained frequently, is computationally expensive due to rebuilding of indexes, yet even for the least training interval we used (10), PIMMLI performs better in the long run than the competing approaches, as observed in cumulative results of figure 3.22.

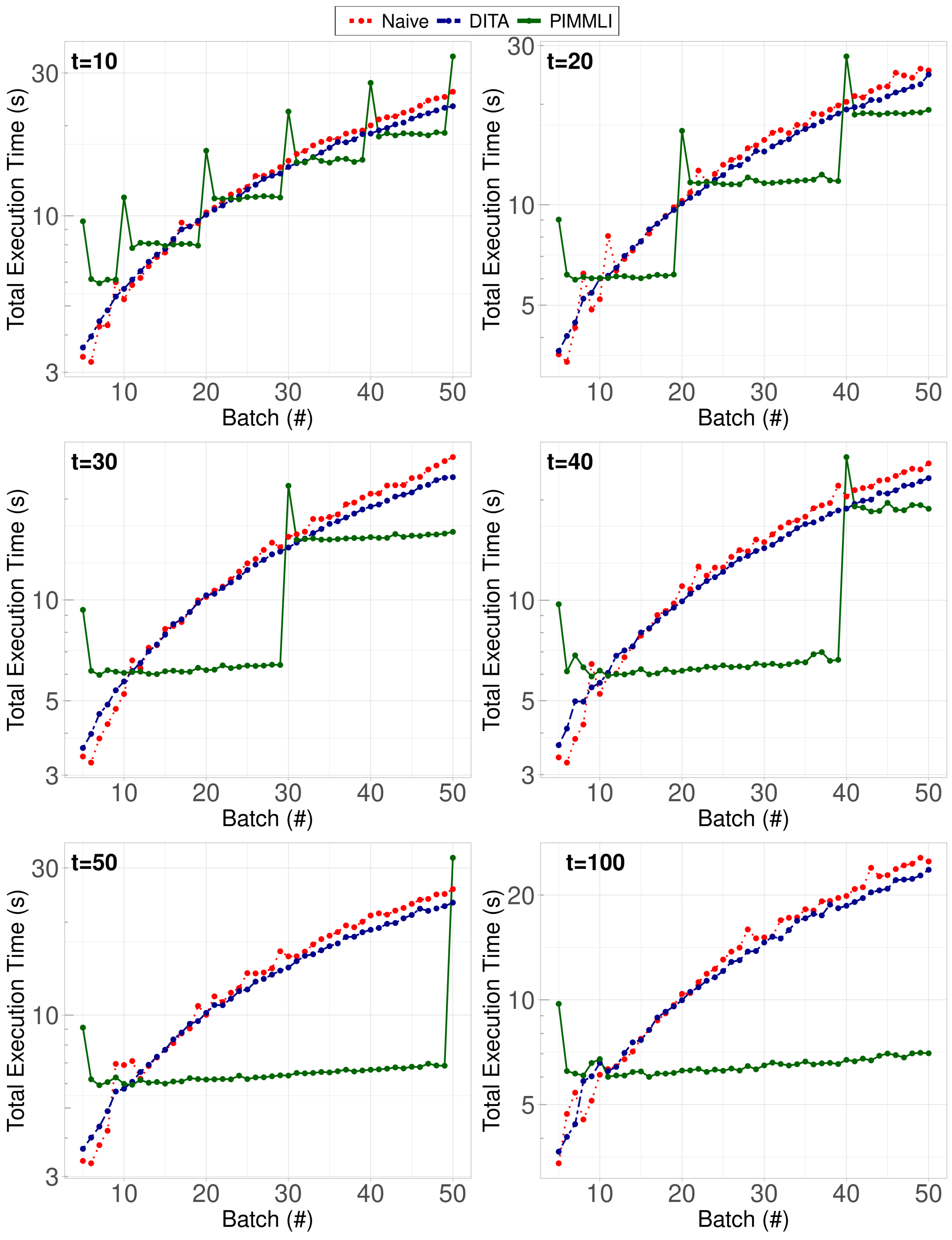


Figure 3.21: Total impact of training interval (t)

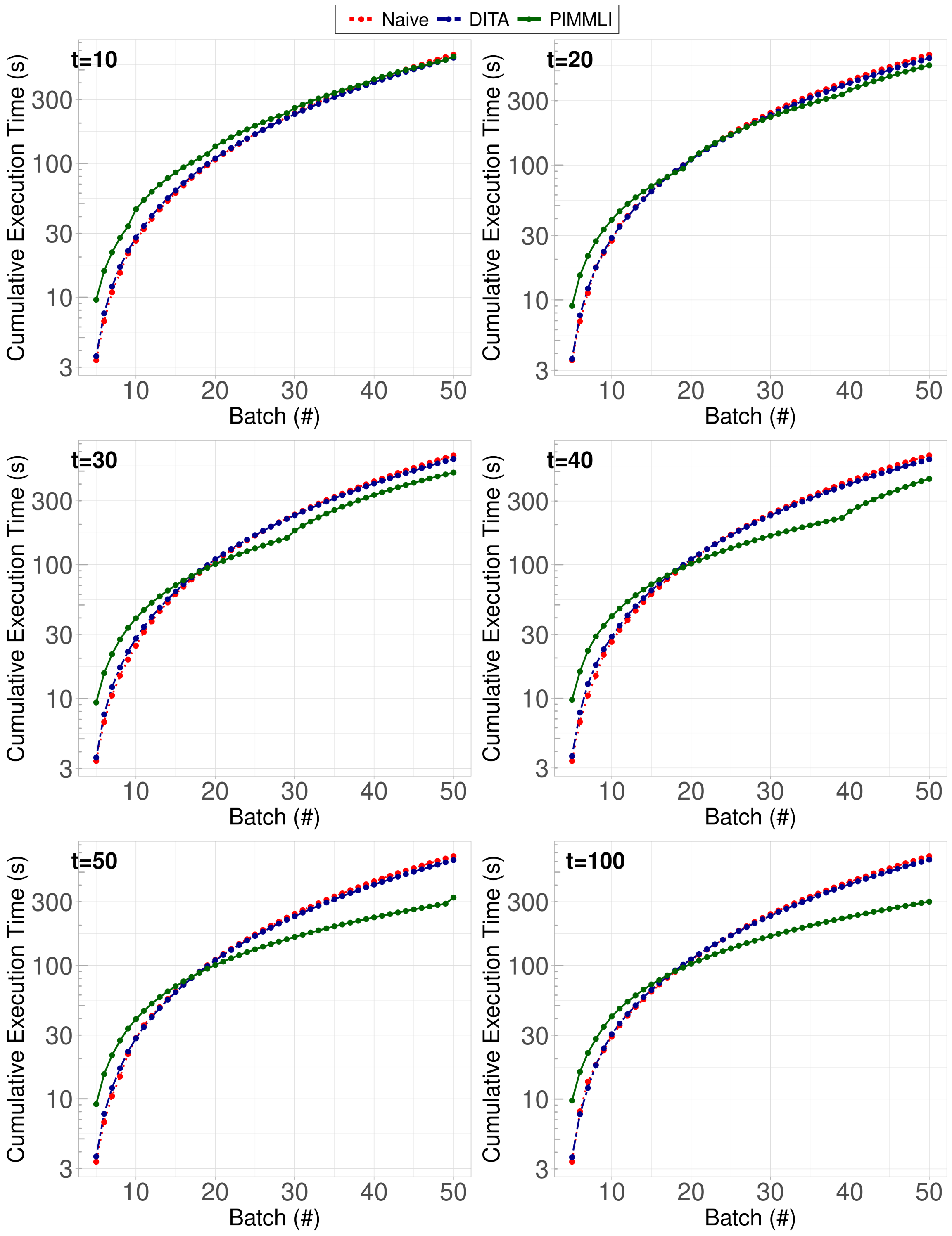


Figure 3.22: Cumulative impact of training interval (t)

4 Conclusions & Future Work

In this chapter, we present our conclusions in Section 4.1 and include future work in Section 4.2 to extend PIMMLI and this research in general.

4.1 Conclusions

- We introduced PIMMLI, the first scalable trajectory stream indexing technique which uses a novel heuristic to foresee future points before indexing them. Foreseeing future points helps ensure that previously built index structures remain valid for several batches in a stream and therefore saving time by not having to rebuild the indexes for each batch. The trade-off of PIMMLI is that it requires a high cost during training intervals but is compensated by reusing existing indexes.

- Our algorithm uses Apache Spark, which is an efficient distributed computing platform built on the same principles as Hadoop but is faster due to in-memory execution. Using the Spark Streaming framework, PIMMLI discretizes trajectory streams in real-time and builds a trainable model for each trajectory. We then use these models to predict future points for each trajectory and index them. For indexing, we use a multi-level approach to facilitate a distributed computing setting. Our algorithm builds a shared global index that helps locate right partitions for a query. We also build local indexes in each partition to locate candidate trajectories. This also helps us in keeping limited and essential data in memory for each node, addressing the memory consumption challenge of Apache Spark. For future batches in each stream, the models are continuously validated in an online fashion, and the indexes are updated if necessary.

- We performed comprehensive experiments against DITA by varying the dataset, systems, and algorithm parameters observing an average of 3.5X improvement in indexing and query execution time of PIMMLI over DITA.

- Our experiments showing the impact of datasets reveal that on certain small and disparate datasets, such as the scaled down version of the Beijing dataset we used, PIMMLI performs upto 19.97X faster than DITA in cumulative indexing time. This factor is decreased as we increase the size and density of the trajectories in the dataset, but still stays more than 3X even in the largest dataset of 12.5 million points with 50 batches of data streams.

- We performed similar experiments against the Naive (non-indexed) approach, where issuing range queries on our resulting indexes performed 34.09X faster than exhaustively searching on streamed trajectories.

- Our results include the total as well as the cumulative execution time performance of PIMMLI and competing approaches. In total execution times, it can be observed that the Naive approach performs better than others when the number of streamed batches is small. This supports the typical conclusion of indexing on less data that the overhead of indexing smaller amounts of data supersedes the relatively smaller benefits it provides in queries later. We also notice in total execution results that PIMMLI pays a higher computational cost at the beginning of the stream but the later batches reap the benefits of the predictive approach, as observed in cumulative results.

4.2 Future Work

- In our experiments, we performed range query processing using PIMMLI and competing approaches. Range Queries are a common type of queries used in real-time analytics, as described in the Background chapter. A possible future research direction is to extend PIMMLI to handle *Join* and *Similarity Search* queries.

- The setup of this research includes streaming points to each trajectory, i.e. we are streaming data and our algorithm later optimizes it for faster query processing. There are a few studies that have studied the case where queries are also streamed. This includes issuance of real-time continuous queries in an online fashion just as we add data points. It may be interesting to observe the performance of such system holistically when both demand and supply of data points simultaneously are at high rate.

- PIMMLI balances data in several partitions using STR partioning as a bulk-loading mechanism for initially built R-trees. To maximize index reusability, the data used in such initial partitioning is predicted by trajectory models. It may be an important future work to keep different partitions balanced for longer-term especially for cases when there is huge variation in the size of stream for each trajectory. It may also be an interesting work to maintain balance of computational loads among several worker nodes.

- It may be interesting to experimentally compare PIMMLI with a very recent technique, Dragoon [27] which supports data-streams and builds indexes using historic trajectory data.

- PIMMLI accepts incoming batches of stream in real-time, train/validate/retrain models of trajectories and builds/updates indexes. Later, the trajectories are stored in the original sequence with exact same number of points as received, with pointers from the indexes. To optimize query times even further, it may be important to remove *less-informative* points. Pruning any points that may not infer particularly useful information related to domain can be one of the ways of reducing the past data stored of continuously evolving trajectories. The benefits will be two-fold: (a) This will require lesser storage resources at each worker node; (b) It will reduce query times as the exhaustive search will be faster among the candidate trajectories located through the index.

References

[1] Debi Prasanna Acharjya and Kauser Ahmed. “A survey on big data analytics: challenges, open research issues and tools”. In: *International Journal of Advanced Computer Science and Applications* 7.2 (2016), pp. 511–518 (cit. on p. 8).

[2] Charu C Aggarwal. *Data streams: models and algorithms.* Vol. 31. Springer Science & Business Media, 2007 (cit. on p. 3).

[3] Louai Alarabi, Mohamed F Mokbel, and Mashaal Musleh. “St-hadoop: A mapreduce framework for spatio-temporal data”. In: *GeoInformatica* 22.4 (2018), pp. 785–813 (cit. on pp. 13, 15).

[4] Shivnath Babu and Jennifer Widom. “Continuous queries over data streams”. In: *ACM Sigmod Record* 30.3 (2001), pp. 109–120 (cit. on p. 3).

[5] Petko Bakalov, Marios Hadjieleftheriou, and Vassilis J Tsotras. “Time relaxed spatiotemporal trajectory joins”. In: *Proceedings of the 13th annual ACM international workshop on Geographic information systems.* 2005, pp. 182–191 (cit. on p. 12).

[6] Petko Bakalov et al. “Efficient trajectory joins using symbolic representations”. In: *Proceedings of the 6th international conference on Mobile data management.* 2005, pp. 86–93 (cit. on p. 12).

[7] Jon Louis Bentley. “Multidimensional binary search trees in database applications”. In: *IEEE Transactions on Software Engineering* 4 (1979), pp. 333–340 (cit. on p. 3).

[8] Jürgen Beringer and Eyke Hüllermeier. “Online clustering of parallel data streams”. In: *Data & knowledge engineering* 58.2 (2006), pp. 180–204 (cit. on p. 3).

[9] Albert Bifet et al. “Streamdm: Advanced data mining in spark streaming”. In: *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE. 2015, pp. 1608–1611 (cit. on p. 8).

[10] Ruichu Cai et al. “DITIR: distributed index for high throughput trajectory insertion and real-time temporal range query”. In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1865–1868 (cit. on pp. 3, 11, 12, 15).

[11] Zhi Cai et al. “Vector-based trajectory storage and query for intelligent transport system”. In: *IEEE Transactions on Intelligent Transportation Systems* 19.5 (2017), pp. 1508–1519 (cit. on p. 3).

[12] Paris Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015) (cit. on p. 6).

[13] V Prasad Chakka, Adam Everspaugh, Jignesh M Patel, et al. “Indexing large trajectory data sets with SETI.” In: *CIDR*. Vol. 75. Citeseer. 2003, p. 76 (cit. on pp. 3, 12).

[14] Lu Chen et al. “Real-time distributed co-movement pattern detection on streaming trajectories”. In: *Proceedings of the VLDB Endowment* 12.10 (2019), pp. 1208–1220 (cit. on p. 14).

[15] S. Chintapalli et al. “Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pp. 1789–1792. DOI: 10.1109/IPDPSW.2016.138 (cit. on pp. 6, 10).

[16] Chi-Yin Chow, Jie Bao, and Mohamed F Mokbel. “Towards location-based social networking services”. In: *proceedings of the 2nd ACM SIGSPATIAL International Workshop on location based social networks*. 2010, pp. 31–38 (cit. on p. 5).

[17] Anna Ciampi, Annalisa Appice, and Donato Malerba. “Summarization for geographically distributed data streams”. In: *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Springer. 2010, pp. 339–348 (cit. on p. 3).

[18] Philippe Cudre-Mauroux, Eugene Wu, and Samuel 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 pp. 3, 15).

[19] Ticiana L Coelho Da Silva, Karine Zeitouni, and José AF de Macêdo. “Online clustering of trajectory data stream”. In: *2016 17th IEEE International Conference on Mobile Data Management (MDM)*. Vol. 1. IEEE. 2016, pp. 112–121 (cit. on p. 3).

[20] Jiafeng Ding et al. “Real-time trajectory similarity processing using longest common subsequence”. In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science

and Systems (HPCC/SmartCity/DSS). IEEE. 2019, pp. 1398–1405 (cit. on p. 14).

[21] Xin Ding et al. “Ultraman: a unified platform for big trajectory data management and analytics”. In: *Proceedings of the VLDB Endowment* 11.7 (2018), pp. 787–799 (cit. on pp. 13, 15).

[22] Pedro Domingos and Geoff Hulten. “Mining high-speed data streams”. In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2000, pp. 71–80 (cit. on p. 3).

[23] Ahmed Eldawy, Louai Alarabi, and Mohamed F Mokbel. “Spatial partitioning techniques in SpatialHadoop”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1602–1605 (cit. on p. 24).

[24] Ahmed Eldawy and Mohamed F Mokbel. “A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data”. In: *Proceedings of the VLDB Endowment* 6.12 (2013), pp. 1230–1233 (cit. on p. 13).

[25] Ahmed Eldawy et al. “Sphinx: Distributed execution of interactive sql queries on big spatial data”. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2015, pp. 1–4 (cit. on p. 13).

[26] Martin Ester, Hans-Peter Kriegel, and Jörg Sander. “Spatial data mining: A database approach”. In: *International Symposium on Spatial Databases*. Springer. 1997, pp. 47–66 (cit. on p. 2).

[27] Ziquan Fang et al. “Dragoon: a hybrid and efficient big trajectory management system for offline and online analytics”. In: *The VLDB Journal* 30.2 (2021), pp. 287–310 (cit. on pp. 14, 15, 60).

[28] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. "Mining data streams: a review". In: *ACM Sigmod Record* 34.2 (2005), pp. 18–26 (cit. on p. 3).

[29] Zdravko Galić et al. "Geospatial data streams: Formal framework and implementation". In: *Data & Knowledge Engineering* 91 (2014), pp. 1–16 (cit. on p. 12).

[30] Joe Grengs, Xiaoguang Wang, and Lidia Kostyniuk. "Using GPS data to understand driving behavior". In: *Journal of urban technology* 15.2 (2008), pp. 33–53 (cit. on p. 2).

[31] Lei Gu and Huan Li. "Memory or time: Performance evaluation for iterative operation on hadoop and spark". In: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE. 2013, pp. 721–727 (cit. on p. 7).

[32] Sudipto Guha and Nina Mishra. "Clustering data streams". In: *Data stream management*. Springer, 2016, pp. 169–187 (cit. on p. 3).

[33] Antonin 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 pp. 5, 12).

[34] Marios Hadjieleftheriou et al. "Complex spatio-temporal pattern queries". In: *VLDB*. Vol. 5. 2005, pp. 877–888 (cit. on p. 3).

[35] Marios Hadjieleftheriou et al. "Indexing spatiotemporal archives". In: *The VLDB Journal* 15.2 (2006), pp. 143–164 (cit. on p. 3).

[36] Yuxing Han et al. “Efficiently retrieving top-k trajectories by locations via traveling time”. In: *Australasian Database Conference*. Springer. 2014, pp. 122–134 (cit. on p. 3).

[37] Akaash Vishal Hazarika, G Jagadeesh Sai Raghu Ram, and Eeti Jain. “Performance comparision of Hadoop and spark engine”. In: *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*. IEEE. 2017, pp. 671–674 (cit. on p. 7).

[38] Benjamin Hindman et al. “Mesos: A platform for fine-grained resource sharing in the data center.” In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22 (cit. on p. 8).

[39] Nicola Hönle et al. “Usability analysis of compression algorithms for position data streams”. In: *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2010, pp. 240–249 (cit. on p. 3).

[40] Jon S Horne et al. “Analyzing animal movements using Brownian bridges”. In: *Ecology* 88.9 (2007), pp. 2354–2363 (cit. on p. 5).

[41] Ling Hu et al. “Spatial query integrity with voronoi neighbors”. In: *IEEE Transactions on Knowledge and Data Engineering* 25.4 (2011), pp. 863–876 (cit. on p. 6).

[42] James N Hughes et al. “Geomesa: a distributed architecture for spatio-temporal fusion”. In: *Geospatial informatics, fusion, and motion video analytics V*. Vol. 9473. International Society for Optics and Photonics. 2015, 94730F (cit. on p. 13).

[43] Geoff Hulten, Laurie Spencer, and Pedro Domingos. “Mining time-changing data streams”. In: *Proceedings of the seventh ACM SIGKDD international*

conference on Knowledge discovery and data mining. 2001, pp. 97–106 (cit. on p. 3).

[44] Muhammad Hussain Iqbal and Tariq Rahim Soomro. “Big data analysis: Apache storm perspective”. In: *International journal of computer trends and technology* 19.1 (2015), pp. 9–14 (cit. on p. 6).

[45] Ji Jin, Ning An, and Anand Sivasubramaniam. “Analyzing range queries on spatial data”. In: *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE. 2000, pp. 525–534 (cit. on p. 5).

[46] Qi Jin. “Techniques for Analyzing Range Queries on R-Trees”. PhD thesis. Pennsylvania State University, 1999 (cit. on p. 5).

[47] Ibrahim Kamel and Christos Faloutsos. “On packing R-trees”. In: *Proceedings of the second international conference on Information and knowledge management*. 1993, pp. 490–499 (cit. on p. 5).

[48] Nikolaos Koutroumanis et al. “NoDA: Unified NoSQL Data Access Operators for Mobility Data”. In: *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*. 2019, pp. 174–177 (cit. on p. 13).

[49] Marius Laska et al. “A scalable architecture for real-time stream processing of spatiotemporal IoT stream data—Performance analysis on the example of map matching”. In: *ISPRS International Journal of Geo-Information* 7.7 (2018), p. 238 (cit. on p. 3).

[50] Alemu Lelago et al. “Assessment and mapping of status and spatial distribution of soil macronutrients in Kambata Tembaro zone, Southern Ethiopia”. In: *Advances in Plants and Agriculture Research* 4.4 (2016), pp. 305–317 (cit. on p. 2).

[51] Scott T Leutenegger, Mario A Lopez, and Jeffrey 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 pp. 24, 36).

[52] Ruiyuan Li et al. "Trajmesa: A distributed nosql storage engine for big trajectory data". In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 2002–2005 (cit. on p. 13).

[53] Shiqiang Li et al. "An Effective Spatio-Temporal Query Framework for Massive Trajectory Data in Urban Computing". In: *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2019, pp. 586–593 (cit. on p. 13).

[54] Jiamin Lu and Ralf Hartmut Güting. "Parallel secondo: Practical and efficient mobility data processing in the cloud". In: *2013 IEEE International Conference on Big Data*. IEEE. 2013, pp. 107–25 (cit. on p. 13).

[55] Qiang Ma et al. "Query processing of massive trajectory data based on mapreduce". In: *Proceedings of the first international workshop on Cloud data management*. 2009, pp. 9–16 (cit. on p. 13).

[56] Altti Ilari Maarala et al. "Low latency analytics for streaming traffic data with Apache Spark". In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE. 2015, pp. 2855–2858 (cit. on p. 10).

[57] Feng Mao, Minhe Ji, and Ting Liu. "Mining spatiotemporal patterns of urban dwellers from taxi trajectory data". In: *Frontiers of Earth Science* 10.2 (2016), pp. 205–221 (cit. on p. 3).

[58] Ilias Mavridis and Helen Karatza. “Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark”. In: *Journal of Systems and Software* 125 (2017), pp. 133–151 (cit. on p. 6).

[59] Mohamed F Mokbel et al. “Continuous query processing of spatio-temporal data streams in place”. In: *GeoInformatica* 9.4 (2005), pp. 343–365 (cit. on p. 12).

[60] Luis Moreira-Matias et al. “Predicting taxi–passenger demand using streaming data”. In: *IEEE Transactions on Intelligent Transportation Systems* 14.3 (2013), pp. 1393–1402 (cit. on pp. 4, 16, 32, 41).

[61] Jonathan Muckell et al. “SQUISH: an online approach for GPS trajectory compression”. In: *Proceedings of the 2nd international conference on computing for geospatial research & applications*. 2011, pp. 1–8 (cit. on p. 3).

[62] Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005 (cit. on p. 3).

[63] Salvatore Orlando et al. “Trajectory data warehouses: design and implementation issues”. In: *Journal of computing science and engineering* 1.2 (2007), pp. 211–232 (cit. on p. 3).

[64] Bernd-Uwe Pagel et al. “Towards an analysis of range query performance in spatial data structures”. In: *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1993, pp. 214–221 (cit. on p. 5).

[65] Zhicheng Pan et al. “TraSP: A General Framework for Online Trajectory Similarity Processing”. In: *International Conference on Web Information Systems Engineering*. Springer. 2020, pp. 384–397 (cit. on pp. 14, 15).

[66] Dimitris Papadias et al. “Query processing in spatial network databases”. In: *Proceedings 2003 VLDB Conference*. Elsevier. 2003, pp. 802–813 (cit. on p. 4).

[67] Kostas Patroumpas and Timos Sellis. “Event processing and real-time monitoring over streaming traffic data”. In: *International Symposium on Web and Wireless Geographical Information Systems*. Springer. 2012, pp. 116–133 (cit. on p. 12).

[68] Guido Proietti and Christos Faloutsos. “I/O complexity for range queries on region data stored using an R-tree”. In: *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*. IEEE. 1999, pp. 628–635 (cit. on p. 5).

[69] Shaojie Qiao et al. “Predicting long-term trajectories of connected vehicles via the prefix-projection technique”. In: *IEEE Transactions on Intelligent Transportation Systems* 19.7 (2017), pp. 2305–2315 (cit. on p. 40).

[70] Jiwei Qin, Liangli Ma, and Qing Liu. “Dfthr: A distributed framework for trajectory similarity query based on hbase and redis”. In: *Information* 10.2 (2019), p. 77 (cit. on p. 13).

[71] Sayan Ranu et al. “Indexing and matching trajectories under inconsistent sampling rates”. In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, pp. 999–1010 (cit. on pp. 3, 12, 15).

[72] Jorge L Reyes-Ortiz, Luca Oneto, and Davide Anguita. “Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf”. In: *Procedia Computer Science* 53 (2015), pp. 121–130 (cit. on p. 6).

[73] Tasneem Salah et al. “The evolution of distributed systems towards microservices architecture”. In: *2016 11th International Conference for Internet Tech-*

nology and Secured Transactions (ICITST). IEEE. 2016, pp. 318–325 (cit. on p. 5).

[74] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. “Dita: Distributed in-memory trajectory analytics”. In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 725–740 (cit. on pp. 3, 11, 13, 15, 16, 32, 35).

[75] Reza Sherkat and Davood Rafiei. “On efficiently searching trajectories and archival data for historical similarities”. In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 896–908 (cit. on p. 3).

[76] Konstantin Shvachko et al. “The hadoop distributed file system”. In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10 (cit. on p. 6).

[77] Keng Siau, Ee-Peng Lim, and Zixing Shen. “Mobile commerce: Promises, challenges and research agenda”. In: *Journal of Database Management (JDM)* 12.3 (2001), pp. 4–13 (cit. on p. 5).

[78] Haoyu Tan, Wuman Luo, and Lionel M Ni. “Clost: a hadoop-based storage system for big spatio-temporal data analytics”. In: *Proceedings of the 21st ACM international conference on Information and knowledge management*. 2012, pp. 2139–2143 (cit. on p. 13).

[79] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-hall, 2007 (cit. on p. 5).

[80] Lu-An Tang et al. “Retrieving k-nearest neighboring trajectories by a set of point locations”. In: *International Symposium on Spatial and Temporal Databases*. Springer. 2011, pp. 223–241 (cit. on p. 3).

[81] Yannis Theodoridis and Dimitris Papadias. “Range queries involving spatial relations: A performance analysis”. In: *International Conference on Spatial Information Theory*. Springer. 1995, pp. 537–551 (cit. on p. 5).

[82] Eleftherios Tiakas et al. “Trajectory similarity search in spatial networks”. In: *2006 10th International Database Engineering and Applications Symposium (IDEAS’06)*. IEEE. 2006, pp. 185–192 (cit. on p. 11).

[83] Angelos Valsamis et al. “Employing traditional machine learning algorithms for big data streams analysis: The case of object trajectory prediction”. In: *Journal of Systems and Software* 127 (2017), pp. 249–257 (cit. on p. 3).

[84] Vinod Kumar Vavilapalli et al. “Apache hadoop yarn: Yet another resource negotiator”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–16 (cit. on p. 8).

[85] Aggelos Vlavianos, Marios Iliofotou, and Michalis Faloutsos. “BiToS: Enhancing BitTorrent for supporting streaming applications”. In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE. 2006, pp. 1–6 (cit. on p. 40).

[86] Mehul Nalin Vora. “Hadoop-HBase for large-scale data”. In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. Vol. 1. IEEE. 2011, pp. 601–605 (cit. on p. 12).

[87] Haozhou Wang et al. “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. 12).

[88] Xiangyu Wang et al. "Search me in the dark: Privacy-preserving boolean range query over encrypted spatial data". In: *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE. 2020, pp. 2253–2262 (cit. on p. 6).

[89] Xiaoqian Wu and Chuanqin Zang. "A new spatial index structure for GIS data". In: *2009 Third International Conference on Multimedia and Ubiquitous Engineering*. IEEE. 2009, pp. 471–476 (cit. on p. 3).

[90] Dong Xie et al. "Simba: spatial in-memory big data analysis". In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2016, pp. 1–4 (cit. on pp. 3, 13).

[91] Guowen Xu et al. "Enabling efficient and geometric range query with access control over encrypted spatial data". In: *IEEE Transactions on Information Forensics and Security* 14.4 (2018), pp. 870–885 (cit. on p. 5).

[92] Munkh-Erdene Yadamjav et al. "Efficient Multi-range Query Processing on Trajectories". In: *International Conference on Conceptual Modeling*. Springer. 2018, pp. 269–285 (cit. on p. 5).

[93] Man Lung Yiu et al. "Enabling search services on outsourced private spatial data". In: *The VLDB Journal* 19.3 (2010), pp. 363–384 (cit. on p. 5).

[94] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. "Geospark: A cluster computing framework for processing large-scale spatial data". In: *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*. 2015, pp. 1–4 (cit. on p. 13).

[95] Haitao Yuan and Guoliang Li. "Distributed in-memory trajectory similarity search and join on road network". In: *2019 IEEE 35th international conference on data engineering (ICDE)*. IEEE. 2019, pp. 1262–1273 (cit. on pp. 11, 13).

[96] Jing Yuan et al. “T-drive: driving directions based on taxi trajectories”. In: *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*. 2010, pp. 99–108 (cit. on pp. 16, 32, 41).

[97] Matei Zaharia et al. “Apache spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65 (cit. on pp. 6, 7, 9).

[98] Matei Zaharia et al. “Fast and interactive analytics over Hadoop data with Spark”. In: *Usenix Login* 37.4 (2012), pp. 45–51 (cit. on pp. 6, 7).

[99] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 15–28 (cit. on p. 7).

[100] Demetrios Zeinalipour-Yazti, Song Lin, and Dimitrios Gunopulos. “Distributed spatio-temporal similarity search”. In: *Proceedings of the 15th ACM international conference on Information and knowledge management*. 2006, pp. 14–23 (cit. on p. 11).

[101] Kai Zhao, Denis Khryashchev, and Huy Vo. “Predicting taxi and uber demand in cities: Approaching the limit of predictability”. In: *IEEE Transactions on Knowledge and Data Engineering* (2019) (cit. on pp. 4, 10).

[102] Da Zheng, Randal Burns, and Alexander S Szalay. “Toward millions of file system IOPS on low-cost, commodity hardware”. In: *SC’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2013, pp. 1–12 (cit. on p. 5).

[103] Yu Zheng. “Trajectory data mining: an overview”. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 6.3 (2015), pp. 1–41 (cit. on p. 3).

[104] Yu Zheng, Xing Xie, Wei-Ying Ma, et al. “Geolife: A collaborative social networking service among user, location and trajectory.” In: *IEEE Data Eng. Bull.* 33.2 (2010), pp. 32–39 (cit. on pp. 2, 3, 16, 30, 40, 41, 52).

[105] Weitao Zou et al. “Strark-H: A Strategy for Spatial Data Storage to Improve Query Efficiency Based on Spark”. In: *International Conference on Algorithms and Architectures for Parallel Processing.* Springer. 2019, pp. 285–299 (cit. on p. 3).