Efficient Path Query Processing over Massive Trajectories on the Cloud

Ruiyuan Li, Sijie Ruan, Jie Bao, *Member, IEEE*, Yanhua Li, *Senior Member, IEEE*, Yingcai Wu, *Member, IEEE*, Liang Hong, *Member, IEEE*, and Yu Zheng, *Senior Member, IEEE*

Abstract—A path query aims to find trajectories passing a given sequence of connected road segments within a time period. It is very useful in many urban applications: 1) traffic modeling, 2) frequent path mining, 3) intersection coordination, and 4) traffic anomaly detection. Existing solutions for path query processing are implemented based on single machines, which are not efficient for the following tasks: 1) indexing large-scale historical data; 2) handling real-time trajectory updates; and 3) processing concurrent path queries from urban data mining applications. In this paper, we design and implement a cloud-based path query processing framework based on Microsoft Azure. We modify existing suffix tree structure to index trajectories using Azure Table. The proposed system consists of two main parts: 1) *back-end processing*, which performs pre-processing (i.e., parsing and map-matching) and index building tasks with a distributed computing platform (i.e., Storm) used to efficiently handle massive real-time trajectory updates; and 2) *query processing*, which answers path queries using Azure Storm to improve efficiency and overcome I/O bottleneck. Extensive experiments are performed based on the real-time taxi trajectories from Guiyang City, the capital of Guizhou Province, China to confirm the system efficiency. We also demonstrate a real deployed traffic analysis system based on our query processing framework.

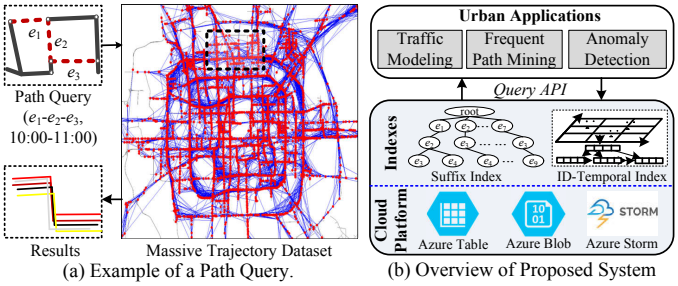
Index Terms—Trajectory Query Processing, Spatio-temporal Data Management, Distributed Computing, Cloud Computing.

1 Introduction

A path query aims to extract qualified trajectories that have passed a user-specified path (i.e., a sequence of connected edges) within a temporal period. Figure 1a gives an example, where a user retrieves the trajectories that have passed edges ($e_1 \rightarrow e_2 \rightarrow e_3$), i.e., the red dotted lines during a time interval (10:00 to 11:00), and the qualified trajectories are returned as the colorful lines.

Many urban applications rely heavily on path queries: 1) traffic speed modeling [1, 2], where extracted trajectories can be used to estimate the travel time/speed of a given path; 2) path mining [3–5], where qualified trajectories can be used for route recommendations; 3) intersection coordination [6], where the trajectories can be used for traffic light coordination; and 4) traffic anomaly detection [7, 8], where we can find anomalous vehicles.

The most straightforward way to tackle this problem is to retrieve all trajectory IDs from an inverted index, i.e., indexing trajectory IDs based on road segment IDs, and then perform a join operation to select qualified results. This naive solution can be extremely inefficient, as it not only needs to scan a large number of candidate trajectories, but also performs join operations that can be very costly. To improve the query efficiency, advanced solutions, e.g., [9, 10], are proposed to use efficient arithmetic operations to verify paths and avoid join operations. There are also solutions, e.g., [1, 11], that take advantage of suffix tree index to speed up join operations. However, these solutions still suffer from three main drawbacks: 1) they still need to maintain a trajectory index (e.g., inverted list or suffix tree) in the memory, which is unrealistic when trajectory data is huge, e.g., millions of trajectories over several years; 2) all of them are implemented based on a single machine, which incurs a performance bottleneck for answering a large number of concurrent path queries from data mining applications; and 3) none of existing solutions is efficient to handle large-scale trajectory updates, which prevents them from providing real-time path query/analysis services.



(a) Example of a Path Query. (b) Overview of Proposed System

Fig. 1: Motivating Examples.

In this paper, we build a cloud-based path query processing system on Microsoft Azure, which extends from our previous work [12]. Figure 1b gives an overview of our system, where we build a layer based on Azure computing and storage components to support both real-time trajectory updates and path queries. A distributed streaming

- *R.Y. Li and S.J. Ruan are with the School of Computer Science and Technology, Xidian University, China. They are now interns at JD Intelligent City Research and JD Urban Computing Business Unit. Partial work of this paper was done when R.Y. Li and S.J. Ruan were interns at Microsoft Research Asia. E-mail: {ruiyuan.li, ruansijie}@jd.com*
- *J. Bao and Y. Zheng are with JD Intelligent City Research and JD Urban Computing Business Unit. Y. Zheng is also affiliated with School of Computer Science and Technology, Xidian University. Jie Bao is the correspondence author of this paper. E-mail: baojie@jd.com and msyuzheng@outlook.com*
- *Y.H. Li is with the Computer Science Department, Worcester Polytechnic Institute, Worcester, MA, USA. E-mail: yli15@wpi.edu*
- *Y.C. Wu is with the State Key Lab of CAD & CG, Zhejiang University, Zhejiang, China. E-mail: ycwu@cad.zju.edu.cn*
- *L. Hong is with the School of Information Management, Wuhan University, Wuhan, China. E-mail: hong@whu.edu.cn*

computing platform, i.e., Azure Storm, is extensively used to overcome the I/O bottleneck both in index building and query processing.

The main idea in our system is to modify the traditional suffix tree index: 1) we set a *max height* to limit the size of suffix tree index; 2) we keep an *hourly count* on each suffix record to indicate data distribution; and 3) we store detailed suffix records on Azure Table to enable parallel I/O access. To answer path queries with a limited height suffix tree index, we propose different heuristics to decompose querying paths, retrieve suffix records from Azure Table, and efficiently reconstruct results.

Our system consists of two main parts: 1) *back-end part*, which receives trajectory updates, and performs trajectory parsing and map-matching tasks. After that, it updates the suffix tree index, stores and organizes trajectory data in Azure Table; and 2) *front-end part*, which receives path queries and utilizes Azure Storm to efficiently retrieve query results. Our main contributions are summarized as follows:

- We build a holistic system based on the cloud computing platform, i.e., Microsoft Azure, to efficiently answer path queries over massive trajectory dataset.
- We develop a *table-based suffix tree index*, with *max height*, *hourly count* and *table storage* to overcome the challenges from indexing and querying massive historical trajectory data.
- We propose an efficient indexing algorithm based on Azure Storm to distribute system I/O overhead and to handle real-time trajectory updates.
- We propose a Storm topology to answer path queries both individually and concurrently. To further reduce response time, different heuristic methods are proposed.
- Extensive experiments are conducted based on real taxi trajectories from Guiyang, the capital of Guizhou Province, China, to demonstrate the efficiency of our system. We also demonstrate a real traffic analysis system based on the query processing system [13].

The remainder of this paper is organized as follows: Section 2 introduces preliminaries, formal problem definition and system overview. Section 3 presents pre-processing module. Index building module is presented in Section 4. Section 5 describes the path query processing module. Experimental results and a demonstration traffic analysis system are given in Section 6. The related works are summarized in Section 8. Finally, Section 9 concludes the paper.

2 Preliminary

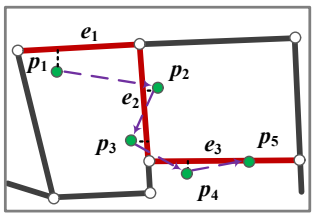
In this section, we first introduce some basic concepts and Azure components used in the paper. After that, we provide a formal definition of trajectory path query and its extensions. Finally, we provide an overview of our cloud-based path query processing system.

2.1 Basic Concepts

Definition 1. (GPS points) A GPS point p_i contains two pieces of information: 1) spatio-temporal information, which includes a pair of latitude and longitude coordinates, and a timestamp; and 2) attributes, which may include speed, direction, and any other information obtained by sensors. An entry in Figure 2a is an example of GPS points, and the green dots in Figure 2b demonstrate GPS point projections on geographic space.

	Spatio-temporal Properties			Attributes	
p_1	lat_1	lng_1	t_1	$speed_1$	dir_1
p_2	lat_2	lng_2	t_2	$speed_2$	dir_2
...	...	...	...	...	...
p_5	lat_5	lng_5	t_5	$speed_5$	dir_5

(a) An Example of Trajectories.



(b) Trajectory Map-Matching.

Fig. 2: An Example of Trajectory Data.

Definition 2. (GPS Trajectory) A GPS trajectory τ contains a list of GPS points ordered by their timestamps. As shown in Figure 2b, on a two-dimensional plane, we can sequentially connect these GPS points into a curve based on their time serials to form a trajectory $\tau = \{p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_5\}$.

Definition 3. (Road Network) A road network can be viewed as a directed graph $G = (V, E)$, where E is a set of road segments and V represents intersections.

Definition 4. (Map-Matched Trajectory) The raw GPS points are mapped onto their corresponding road network. Figure 2b gives a map-matching example, where the green dots (GPS points) are projected onto corresponding road segments (in red). Thus, the trajectory is converted to $Tr = \{(e_1, t_1) \rightarrow (e_2, t_2) \rightarrow (e_3, t_3)\}$, where e_i is the edge ID and t_i is corresponding timestamp. More details about converting a GPS trajectory to a map-matched trajectory can be found in [14].

Definition 5. (Path) A path is a set of connected road segments $P = \{e_i \rightarrow e_j, ..., \rightarrow e_k\}$, where the order of edges indicates the travel sequence. The consecutive road segments in a path should be connected on the road network G. We define the number of edges in a path as its length. For example, the red lines in Figure 2b form a path of $e_1 \rightarrow e_2 \rightarrow e_3$ with a length of 3.

2.2 Azure Preliminary

We give some brief descriptions here to introduce the main components of Microsoft Azure, which are used in our system: 1) Azure Storage, and 2) Azure HDinsight.

2.2.1 Azure Storage

Azure Storage is a massively scalable cloud storage solution[1] with many different components, e.g., Azure Blob, Azure Table, Azure Queue, and Azure Redis.

Azure Blob. Blob (A Binary Large Object) is a collection of binary data stored as a single entity in the Azure storage system. It is equivalent to binary files in a conventional file system. As it can be loaded onto the memory very efficiently, Azure Blob is used to store the index files (i.e. hourly counts) in our system.

Azure Table. Azure Table is a NoSQL database in Azure. Table storage is a key/attribute store with a schema-less

1. https://docs.microsoft.com/en-us/azure/storage/

design. Each storage account can have unlimited number of tables distinguished by table names. Each table entity is identified by two keys: `PartitionKey` and `RowKey`. Table entities with the same `PartitionKey` are stored in the same physical location and can be operated in a batch. Azure Table is very efficient in answering the range queries of `RowKey` within the same `PartitionKey`. To this end, Azure Table is used extensively to store trajectories (more details in [15]).

Azure Queue. Azure Queue provides a reliable messaging solution for asynchronous communication between different application components. In our system, Azure Queue is used as a API channel, which receives/answers API calls from different users/applications.

Azure Redis. Azure Redis, based on the popular open-source Redis cache[2], is an advanced in-memory key-value store. It supports many data structures such as strings, hashes, lists, sets, and sorted sets. To ensure data consistency, Redis supports a set of atomic operations on these data types. As Azure Redis stores data in the memory, it is a good option to put frequently accessed and shared data in our system, e.g., intermediate processed trajectory data and path query results.

2.2.2 Azure HDinsight.

Azure HDinsight[3] is a distributed computing component in Microsoft Azure to perform large-scale data processing, which includes Azure Hadoop, Azure Spark and Azure Storm. To cope with real-time trajectory updates, we use Azure Storm in our system, for its abilities to perform real-time streaming data and online services.

Azure Storm. Azure Storm is a distributed, real-time event processing solution for large, fast streams of data. It is a good choice for processing real-time data and providing online services. There are two types of components in a Storm system: 1) *Spout*, which continuously reads updates/new requests from a message queue (e.g., Azure Queue), and distributes them; and 2) *Bolt*, which is a processing unit. In a Storm program, there will be different kinds of *bolts* with different functions. *Bolts* get tasks from *spouts* and are connected to each other based on the design of users, forming a *Storm Topology*. In our system, we adopt Azure Storm to perform index building and path query processing to overcome I/O issues.

2.3 Problem Definition

Path Query. The path query we address in this paper can be formalized as follows: given a path with a list of connected edges $P = \{e_i, e_j, ..., e_k\}$, a temporal range with a start time T_s and an end time T_e, and a map-matched trajectory dataset $T = \{Tr_1, Tr_2, ..., Tr_n\}$, we want to find all sub-trajectories of Tr_i in T, where Tr_i passed the path P within the given temporal period, i.e., $\{(e_i, t_i), (e_j, t_j), ..., (e_k, t_k)\} \in Tr_i$ and $t_i \geq T_s$ & $t_k \leq T_e$. The objective here is to improve efficiency.

Extension: Partial Path Query. We can consider the path query as a building block to answer a complex partial path query, where the path in a query is not fully connected, but with a set of disconnected road segments. For example, a partial query path $P = \{(e_1, e_2, e_3); (e_i, e_{i+1}, e_{i+2}); (e_j, e_{j+1}, e_{j+2})\}$ contains three sets of road segments (grouped by the parenthesis), where the edges in the same group are connected.

2. https://azure.microsoft.com/en-us/services/cache/
3. https://azure.microsoft.com/en-us/services/hdinsight/

2.4 System Overview

Figure 3 gives an overview of our cloud-based path query processing system, which includes two main components:

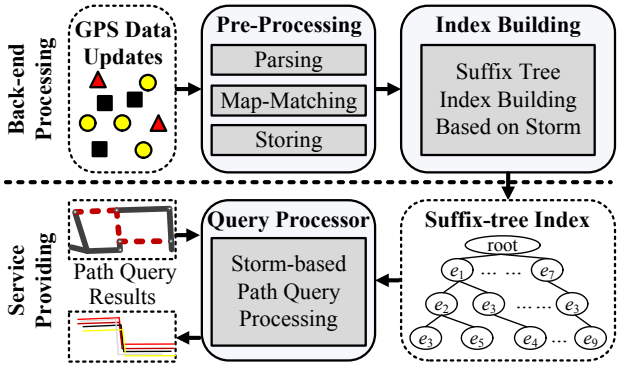


Fig. 3: System Overview.

Back-end Processing. This component, running at the back-end of our system, receives GPS updates and builds trajectory index, as illustrated in the upper part of Figure 3. This component contains two modules: 1) *Pre-Processing*, which gets raw GPS updates, and performs parsing, map-matching, and storing tasks (Detailed in Section 3); and 2) *Index Building*, which builds suffix tree index to speed up path query processing (Detailed in Section 4).

Service Providing. This is a front-end processing component that answers path queries, as illustrated in the bottom part of Figure 3. The main module is *Query Processor*, which takes advantage of the suffix tree index and employs the Storm parallel computing platform to answer both path queries and partial queries in an efficient way (Detailed in Section 5).

3 Trajectory Pre-processing

In this section, we describe the main steps in *pre-processing* module, as shown in Figure 4, which consists of three main steps:

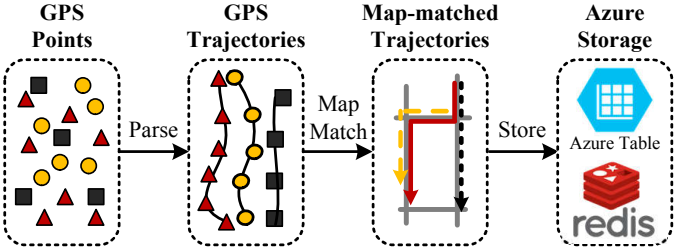


Fig. 4: Overview of Trajectory Pre-processing Module.

Step 1. Trajectory Parsing. As illustrated in Figure 4, in order to support online query processing, our system continuously gets GPS updates from environment, e.g., taxis or personal vehicles. The first step filters noisy GPS points, and

groups GPS points based on their IDs (e.g., plate numbers). We use a heuristics-based outlier detection method in [16].

Step 2. Map-Matching. In this step, the system takes cleaned trajectories and road networks, and then maps each GPS point onto its corresponding road segment. In this step, we use an interactive-voting based map matching-algorithm [14] to perform the map-matching task. Moreover, to support real-time services, we deploy the map-matching algorithm in Storm to expedite the process, with more details can be found in our previous work [15].

Step 3. Storing. As depicted in Figure 4, after performing the map-matching task, we store the processed trajectories in two locations: 1) Azure Table, where each trajectory is distinguished by its ID (as the table name). The temporal ranges are used as `PartitionKey` and exact timestamps are used as `RowKey`. The detailed design of keys can be found in our previous work [17]. In this way, we can efficiently answer ID-temporal queries (i.e., finding the sub-trajectories of a specified moving object within a given time period); and 2) Azure Redis, which is used to cache the map-matched trajectory data for the index building module.

4 Index Building

In this section, we first present the existing suffix tree index structure, which is widely used to answer trajectory path queries. After that, we highlight the major drawbacks if we adopt original suffix tree index directly, and present our table-based suffix tree index. Then, we develop a basic procedure to build table-based suffix tree index. Finally, we describe our Strom-based implementation for building table-based suffix tree index to overcome the drawbacks in the basic index building algorithm.

4.1 Index Structure

4.1.1 Original Suffix Tree Index

Suffix tree index is originally used to index strings [18], which improves the performance of string suffix search. In the case of trajectory data management, a map-matched trajectory can be considered as a string, where each edge ID is equivalent to a character and a path query can be mapped as a string suffix search (i.e., searching by a sequence of edge IDs is similar to searching by a sequence of characters). It has been demonstrated in many existing systems, e.g., [1, 11], that suffix tree index is a very efficient solution to answer path queries.

Figure 5 gives an example of original suffix tree index structure. In this example, a trajectory Tr_1 passes four edges (i.e., e_1, e_2, e_3 and e_4). As a result, four suffixes are generated based on this trajectory, i.e., (e_1, e_2, e_3, e_4); (e_2, e_3, e_4); (e_3, e_4); (e_4). Then, all four suffixes are inserted into the corresponding positions (marked in shade) on the suffix tree index.

With the suffix tree index, query processing becomes straightforward: when a user asks for the trajectories that passed a path, the system just traverses to the corresponding node in the suffix tree index and retrieves all entries associated with it. For example, if a user issues a path query with $P = \{e_1, e_2\}$, we can start from the root, and traverse through node e_1 and e_2. All the items in orange color are the query results, i.e., sub-trajectories of Tr_1 and Tr_2.

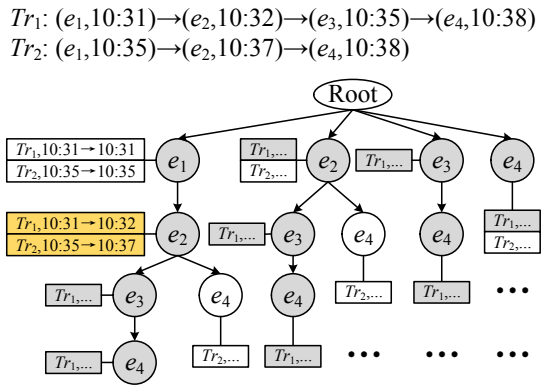


Fig. 5: Original Suffix Tree Index.

However, as illustrated in Figure 5, the original suffix tree index needs to take a lot of space to store the information of trajectory suffixes, as it generates a lot of suffixes even for one trajectory. Moreover, the size of suffix tree index increases significantly when the trajectory dataset is huge, e.g., millions of historical trajectories. Furthermore, the original suffix tree index is not optimized for temporal predicate. Retrieving the qualified trajectories within a given temporal period based on the original suffix tree index incurs significant I/O overhead in accessing a lot of disqualified data. As a result, existing systems [1, 11] only use the suffix tree index to hold the most frequent or most recent trajectories, and cannot support trajectory path queries over entire historical dataset or with a temporal constraint.

4.1.2 Table-based Suffix Tree Index

To efficiently support trajectory path queries at large-scale trajectories, i.e., with months and years of trajectory data, we modify the original suffix tree index. Figure 6 gives an example of our proposed table-based suffix tree index, which consists of two main components: 1) suffix tree index, which includes a tree structure and a set of statistics that are stored in Azure Blob and loaded onto memory during query processing; and 2) suffix records, which store actual trajectories organized based on their suffixes in Azure Table. There are three changes comparing to the traditional suffix tree index:

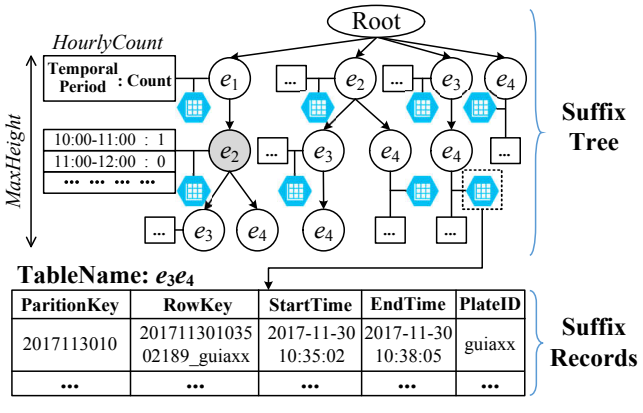


Fig. 6: Table-based Suffix Tree Index.

•*Max Height*. We set a *max height* H to limit the total size of a suffix tree. In other words, the suffix tree only holds the information of suffixes with a length of no more than

H. In this way, we can guarantee that the index can fit in the memory regardless of the size of trajectory data, as it is bounded by the number of sub-paths with *max height* edges. Take Figure 6 as an example, the *max height* of suffix tree index is three. Choosing a suitable *max height* in a system is a design trade-off: if the *max height* is small, it favors the index building, as fewer suffixes are generated. However, it hurts the efficiency in query processing, as more join operations and data accesses are introduced in the query processing phase (more details in Section 6).

• *Hourly Count.* In each node of suffix tree, we maintain a dictionary, where we divide a day into 24 hours and store the average number of trajectories passing the node (i.e., the sequence of edges) in each hour. In the upper left of Figure 6, we present an example for the node e_1e_2 (marked in shade). *Hourly count* serves as a hint for query processing module to make an efficient and smart query plan, with which less data will be accessed (see Section 5.1.1).

• *Table Storage.* As demonstrated in Figure 6, each node also keeps a pointer to an Azure table, where the actual trajectory data is stored. The name of a table is the sub-path ID (table e_3e_4 in the example). The `PartitionKey` of Azure Table is the temporal range, e.g., by hour in the figure, and the `RowKey` is the timestamp with trajectory ID, which can avoid key conflicts in a table when two trajectories of different moving objects are generated at the same time. In this way, trajectories passing the same path within the same temporal period are stored in the same Azure Table partition for more efficient access.

Essentially, we store an extra and re-organized copy of trajectory dataset. The main intuition is to store trajectory data in the same table partition, if they passed the same path within the same temporal period. It is different from traditional indexing methods, where the index just keeps pointers to actual records. However, it is a more economic and efficient choice in Azure to store an extra copy, as the storage cost is much cheaper than the computing cost, e.g., it is only about 10 USD per 1TB/month for Azure Storage[4].

4.2 Index Construction

4.2.1 Table-based Suffix Tree Indexing

There are three main steps in constructing a table-based suffix tree index:

Step 1. Suffix Generation. In this step, our system generates suffixes with a length of no more than H from original trajectories. Essentially, we truncate raw trajectories using a sliding window with a window size of H and a step size of one. For example, if we set the *max height* H as two, the map-matched trajectory Tr: $e_1 \rightarrow e_2 \rightarrow e_3$ will be broken into five sub-trajectories: e_1, e_2, e_3, $e_1 \rightarrow e_2$, and $e_2 \rightarrow e_3$.

Step 2. Index Update. In this step, we group the suffixes generated by the previous step and update the *hourly count* at each node of suffix tree index. For example, we group all sub-trajectories that passed $e_1 \rightarrow e_2 \rightarrow e_3$ within 10:00 PM to 11:00 PM, and then update the *hourly count* at the corresponding node (i.e., 10:00 PM-11:00 PM of $e_1e_2e_3$) in the index.

Step 3. Record Insertion. In this step, we insert the grouped sub-trajectories into Azure Table. Sub-trajectories with the same suffix are inserted to the same table, and sub-trajectories generated within the same time period are inserted into the same partition.

The aforementioned steps are very straightforward. However, there are some potential performance bottlenecks: 1) generating trajectory suffixes takes a lot of time, when the trajectory data is huge (i.e., with a large number of moving objects or a long time period); 2) the volume of generated suffix data can be huge, as it essentially creates H times more data than the map-matched trajectories; 3) it generates a large number of different suffixes, where each of them incurs one Azure Table insertion. The insertions to different tables/partitions can only be performed individually, which introduces significant I/O overhead.

4.2.2 Storm-based Indexing Implementation

To overcome the bottlenecks in building a table-based suffix tree index, we develop an index building process based on a distributed streaming environment, i.e., Azure Storm. It is a quite intuitive choice, as: 1) Storm is natural to support continuous trajectories updates that have a lot of similarity with streaming data; 2) it is more efficient to distribute suffix generation operations on multiple computing nodes; 3) with more nodes in Storm, we can overcome the system I/O bottleneck by writing data in parallel to Azure Table.

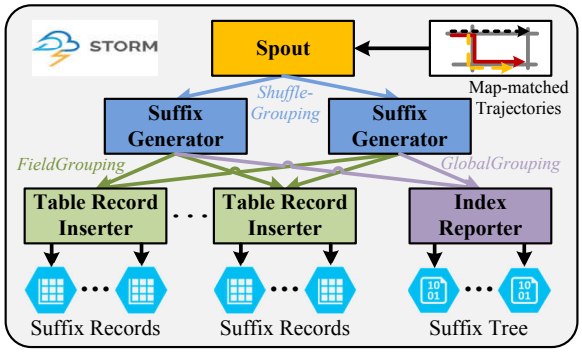


Fig. 7: Overview of Storm Indexing Topology.

Figure 7 gives an overview of the Storm-based Indexing Topology, where the Storm topology is essentially a computing network with four types of components:

• **Spout.** The spout takes map-matched trajectories from Azure Redis, and distributes them to different *Suffix Generator Bolts* using `ShuffleGrouping` mechanism, i.e., random assignment, to achieve a workload balance (shown as blue lines).

• **Suffix Generator Bolt.** This bolt breaks map-matched trajectories into suffixes with the maximum length of H. Then, the generated suffixes are distributed to two locations: 1) *Table Record Inserter Bolt*, using `FieldGrouping` mechanism (as green arrows), i.e., the same suffixes are emitted to the same bolt; and 2) *Index Reporter Bolt*, using `GlobalGrouping` mechanism (as purple arrows), i.e., all suffixes are emitted to one bolt to update the *hourly counts* of *suffix tree index*.

4. https://azure.microsoft.com/en-us/pricing/details/storage/

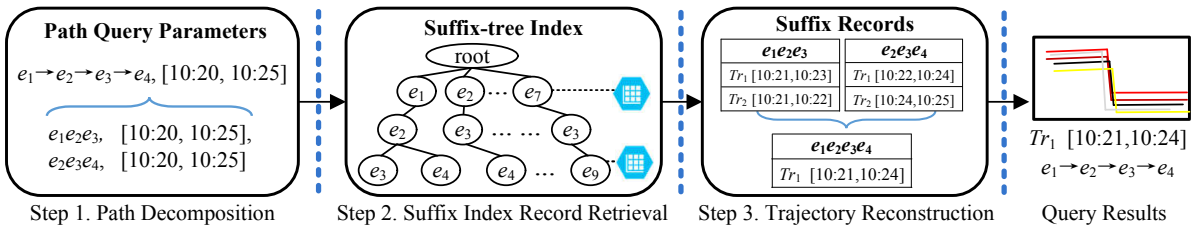


Fig. 8: Overview of Path Query Processing Module.

• **Table Record Inserter Bolt.** This bolt inserts sub-trajectories into Azure Table in batches. It groups sub-trajectories based on their start time, where all sub-trajectories with the same start time period (i.e., in the same hour in our implementation) are inserted into Azure Table as one batch to avoid small writings and overcome I/O bottlenecks.

• **Index Reporter Bolt.** This bolt aggregates all suffixes from *Suffix Generators* and updates the suffix tree index: 1) adding new branches, if a new suffix is generated; or 2) updating the statistics (i.e., *hourly count*) on each node based on new trajectories. Once the index is updated, the new index structure will be stored in an Azure blob.

5 Query Processor

Because the *table-based suffix tree index* employs a *maximum height* H, the qualified trajectories cannot be directly retrieved from the index if the number of edges in a querying path is greater than H. To this end, we propose a new process to answer path queries and implement it in Azure Storm to improve its efficiency. In this section, we first introduce an overall process to answer path queries using a table-based suffix tree index. After that, we present our Storm-based implementation to answer path queries with a better efficiency and overall throughput. Finally, we present an extended Storm topology to answer partial path queries.

5.1 Overall Path Query Process

Figure 8 gives an overview of the querying process with three main steps: 1) *Path Decomposition*, where we decompose the querying path P into several sub-paths; 2) *Suffix Index Record Retrieval*, where we retrieve candidate sub-trajectories from suffix records using the decomposed sub-paths and querying time period; and 3) *Trajectory Reconstruction*, where we reconstruct candidate trajectories based on the retrieved sub-trajectories. Finally, the reconstructed trajectories are returned as results.

5.1.1 Path Decomposition

In this step, we break the querying path P into multiple sub-paths with the maximum length of H (i.e., $P \Rightarrow \{p_1, p_2, ..., p_l\}$, where p_i can be mapped as a table in the suffix tree index, e.g., $e_1 \rightarrow e_2 \rightarrow e_3$).

As shown in Figure 8, the decomposed sub-paths are used to retrieve candidate sub-trajectories from the *table-based suffix tree index*. This process incurs accesses to storage components (i.e., Azure Table), and there are multiple ways in decomposing a path into sub-paths with a length no more than H. Therefore, choosing a suitable way to decompose the querying path is essential to improve the efficiency of query processing. In order to effectively take advantage of the indexed suffix record in Azure Table, the following three requirements should be enforced:

(1) $\forall p_i \in P, |p_i| = H$, i.e., the length of each decomposed sub-path should be equal to the *maximum height* H, as it returns the fewest qualified candidates from Azure Table. Otherwise, for example, when a querying path is decomposed into multiple sub-paths with a length less than H, each sub-path retrieves more disqualified candidates, and it requires more I/O overhead to retrieve them and more computation overhead to join them.

(2) $P = \bigcup_{i=1}^{l} p_i$, which means all edges in the querying path P should be covered in a union set of decomposed sub-paths, as fewer candidates are returned. Otherwise, if we only cover partial path (e.g., $\{e_1 \rightarrow e_2\}$ in $\{e_1 \rightarrow e_2 \rightarrow e_3\}$), candidate trajectories returned by the index will include disqualified trajectories that do not exactly traverse the querying path (e.g., $\{e_1 \rightarrow e_2 \rightarrow e_4\}$).

(3) $\forall p_i, p_{i+1} \in P, p_i \cap p_{i+1} \neq \emptyset$, i.e., there is at least one overlapped edge between every pair of consecutive sub-paths. The main reason here is to avoid additional I/O overhead to access map-matched trajectory data in Azure Table. If there is no overlapped edge between two consecutive decomposed sub-paths (e.g., $P = \{e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_5 \rightarrow e_6\} \Rightarrow [p_1 = \{e_1 \rightarrow e_2 \rightarrow e_3\}, p_2 = \{e_4 \rightarrow e_5 \rightarrow e_6\}]$), the results will contain disqualified trajectories that do not pass the sub-paths p_1 and p_2 consecutively and sequentially (e.g., a trajectory passes p_1, e_7, and then p_2). It requires an additional access to the Azure storage to verify the correctness, which can be very time-consuming.

As each sub-path contains different numbers of trajectories during the querying temporal period, e.g., more in downtown areas, and fewer in suburbs. Different decomposed sub-path combinations retrieve different numbers of trajectories during the querying process, which has significant impact on the query response time. As *hourly count* in the suffix tree index provides an approximate distribution of the number of trajectories on each sub-path, we develop three heuristics to decompose a querying path:

Solution 1. Minimize Sub-path Number. This method uses a sliding window, with a window size of H and a step size of $H - 1$, to decompose the querying path. The main intuition behind this method is to minimize the total number of decomposed sub-paths. Essentially, it aims to minimize the total number of sub-trajectory retrieval queries to the *table-based suffix tree index*.

For example, assuming we have a *table-based suffix-tree* with a *max height* of 3, and a querying path $P = \{e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_5\}$, the querying path P is decomposed into two sub-paths: $p_1 = \{e_1 \rightarrow e_2 \rightarrow e_3\}$ and $p_2 = \{e_3 \rightarrow e_4 \rightarrow e_5\}$.

Solution 2. Minimize Total Count. This method de-

Suffix	$e_1e_2e_3$	$e_2e_3e_4$	$e_3e_4e_5$	$e_4e_5e_6$	$e_5e_6e_7$	$e_6e_7e_8$
Count	60	80	150	85	1	70

(a) Hourly Count in Suffix Tree

i	$f(i)$	u
4	$c(e_1:e_3) + c(e_2:e_4) = 140$	3
5	$c(e_1:e_3) + c(e_3:e_5) = 210$	3
6	$min\begin{cases}f(4)\\f(5)\end{cases} + c(e_4:e_6) = 225$	4
7	$min\begin{cases}f(5)\\f(6)\end{cases} + c(e_5:e_7) = 211$	5
8	$min\begin{cases}f(6)\\f(7)\end{cases} + c(e_6:e_8) = 281$	7

(b) Minimize Total Count

i	$f(i)$	u
4	$max(c(e_1:e_3), c(e_2:e_4)) = 80$	3
5	$max(c(e_1:e_3), c(e_3:e_5)) = 150$	3
6	$max\left(min\begin{cases}f(4)\\f(5)\end{cases}, c(e_4:e_6)\right) = 85$	4
7	$max\left(min\begin{cases}f(5)\\f(6)\end{cases}, c(e_5:e_7)\right) = 85$	6
8	$max\left(min\begin{cases}f(6)\\f(7)\end{cases}, c(e_6:e_8)\right) = 85$	6

(c) Minimize Max Count

Fig. 9: Examples of Querying Path Decomposition Methods.

composes the querying path into sub-paths with a goal of minimizing the total number of *hourly count*. The intuition here is to reduce the size of data (i.e., the total number of sub-trajectories) retrieved from the storage.

We developed a dynamic programming algorithm to find the optimal decomposition method with the minimum total count, of which the state transfer equation is listed as follows:

$$f(i) = \begin{cases} c(e_1 : e_i) & \text{if } 0 < i \leq H \\ \min\limits_{j=1...H-1} f(i-H+j) + c(e_{i-H+1} : e_i) & \text{if } i > H \end{cases}$$

where $f(i)$ denotes the minimum total count when a path is ended with the i^{th} edge e_i, and $c(e_i : e_j)$ denotes the *hourly count* for the corresponding path $e_i \rightarrow e_{i+1} \rightarrow ... \rightarrow e_j$. When the length of querying path is larger than H, its last suffix (i.e., $e_{i-H+1} : e_i$) is always selected (i.e., to fulfill the requirements 1&2). We also use an array $\boldsymbol{u}$ to record the optimal decomposition slot in each step. As a result, the algorithm scans the edges in querying path from back to front and returns decomposed results with a time complexity of $\mathcal{O}(n)$, where n is the length of querying path.

Figure 9 gives an example of decomposing a query path $P = \{e_1 \rightarrow ... \rightarrow e_8\}$, where the *hourly count* of each sub-path is in Figure 9a and the decomposing steps are presented in Figure 9b (column i indicates the step, $f(i)$ calculates the candidate total counts in each step, and u shows the optimal decomposing position in each step). As a result, the querying path is decomposed into four sub-paths: $p_1 = \{e_1 : e_3\}$, $p_2 = \{e_3 : e_5\}$, $p_3 = \{e_5 : e_7\}$, and $p_4 = \{e_6 : e_8\}$ with a total *hourly count* of 281.

Solution 3. Minimize Max Count This method aims to minimize the max number of *hourly count* in each sub-path. The main intuition of this method is to minimize the effect of the worst case scenario (i.e., retrieving a lot of sub-trajectories in some sub-paths, such as edges in downtown areas). It is especially useful in a distributed computing environment, where the total processing time depends on the last finished sub-task.

We apply a similar dynamic programming algorithm to find a path decomposition plan that minimizes their max count. Finding the optimal decomposition tries to combine the previous optimal decomposition with the last edge group, and keep track of their minimum max count. After that, the maximum value between the count of the last suffix and the previous minimum max count is kept. The state transfer equation is defined as follows:

$$f(i) = \begin{cases} c(e_1 : e_i) & \text{if } 0 < i \leq H \\ \max(\min\limits_{j=1...H-1} f(i-H+j), c(e_{i-H+1} : e_i)) & \text{if } i > H \end{cases}$$

where $f(i)$ denotes the minimum max count when the path is ended with the i^{th} edge e_i. In each step, we keep an array $\boldsymbol{u}$ to record the decomposition slot. The complexity of this algorithm is also linear to the length of querying path.

Figure 9a & 9c gives an example of decomposing the querying path using *Minimizing Max Count* method, where the path P is decomposed into four sub-paths $p_1 = \{e_1 : e_3\}$, $p_2 = \{e_2 : e_4\}$, $p_3 = \{e_4 : e_6\}$, and $p_4 = \{e_6 : e_8\}$. Comparing to *Minimize Total Count* method (i.e., $c(p_2) = 150$ & $c(p_3) = 1$), this method (i.e., $c(p_2) = 80$ & $c(p_3) = 85$) returns a more total count (i.e., 165 vs. 151) but with a less variance (i.e., 80 & 85 vs. 150 & 1).

5.1.2 Suffix Index Record Retrieval

In this step, the system retrieves sub-trajectories from Azure Table based on a decomposition plan. For each decomposed sub-path, a temporal range query is issued, where the `Table Name` is a sequence of edge names (e.g., $e_1e_2e_3$), the `PartitionKey` is the temporal information to hours (e.g., 2017122501 for the querying time of 2017/12/25 01:00 - 2017/12/25 01:59), and the `RowKey` is the detailed temporal range (e.g., 201712250100 to 201712250159). If the querying time overlaps with multiple partitions (i.e., covering multiple hours), multiple queries with different `PartitionKeys` are generated.

As a result, each query to the *table-based suffix index* returns a set of sub-trajectories, which consists of trajectory ID, a sequence of edges, and a pair of start/end time period of sub-trajectories. Because this step involves heavy interaction with Azure Storage, it dominates the response time and is potentially a system bottleneck.

5.1.3 Trajectory Reconstruction

In this step, we reconstruct qualified trajectories by joining the sub-trajectories retrieved in the previous step. The trajectory reconstruction process is executed based on the order of decomposed sub-paths. The sub-trajectories traversing the first sub-path is our candidate set. For each sub-trajectory in the candidate set, if it does not appear in all latter sub-paths, the trajectory is discarded. Otherwise, we check if these sub-trajectories have correct overlapped time period between their start and end timestamps. For example, as shown in Step 3 of Figure 8, the querying path $P = \{e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4\}$ is decomposed into $p_1 = \{e_1 : e_3\}$ and $p_2 = \{e_2 : e_4\}$, and the suffix records of both sub-paths contain Tr_1 and Tr_2. However, Tr_1's temporal window in $e_1e_2e_3$ overlaps with that in $e_2e_3e_4$ correctly, while Tr_2's does not. As a result, Tr_1 is a qualified trajectory for this query, while Tr_2 is not.

5.2 Storm-based Path Query Implementation

To overcome the I/O bottleneck in path query processing and efficiently support large-scale concurrent path queries from traffic pattern analysis applications, e.g., [1, 19], we implement our query processing component using Storm.

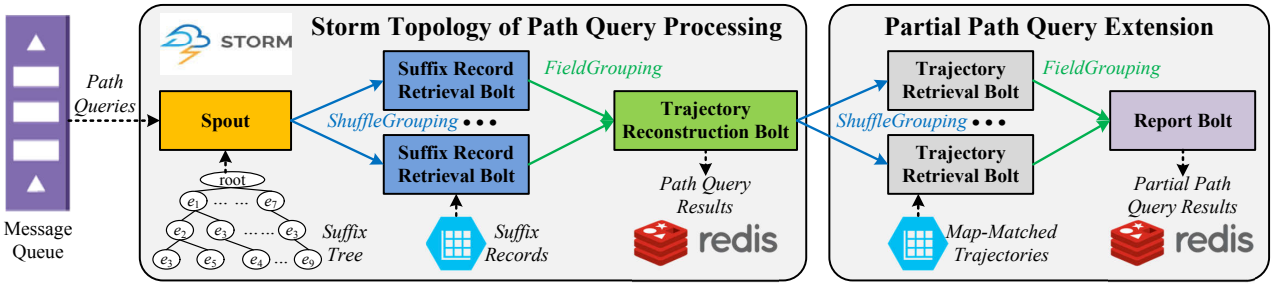


Fig. 10: Storm-based Path Query Processing Topology.

The left portion of Figure 10 depicts the Storm topology with the following three main modules:

• **Spout.** The *spout* has two tasks: 1) it reads querying parameters from an Azure queue (which is used to cache concurrent queries) and assigns IDs to different path queries; and 2) it decomposes querying paths into multiple sub-paths based on the *hourly count* with different heuristics and emits different sub-paths with their query IDs to *Suffix Record Retrieval Bolts* using *ShuffleGrouping* mechanism (i.e., randomly assignment) to balance workloads.

The reasons of putting these two tasks together in one *spout* rather than separately are that: 1) the two tasks are not computationally and I/O intensive, and they are not the bottleneck of path query processing; and 2) the unnecessary network communication cost between different Storm components can be reduced.

• **Suffix Record Retrieval Bolt.** This type of bolt gets decomposed sub-paths and retrieves sub-trajectories from Azure Table. The retrieved sub-trajectories are then emitted to *Trajectory Reconstruction Bolts* using *FieldsGrouping* mechanism based on their query IDs (i.e., the sub-trajectories from the same path query are passed to the same bolt). Distributing suffix record retrieval processing across multiple bolts avoids the system bottleneck of accessing multiple Azure tables and increases the system throughput in dealing with concurrent queries.

• **Trajectory Reconstruction Bolt.** This type of bolt receives sub-trajectory information with the same query ID, and reconstructs overlapped sub-trajectories for each path query. Finally, the bolt writes qualified trajectories as query results to an Azure Redis cache.

5.3 Partial Path Query Extension

For partial path queries, the detailed trajectory information cannot be returned by the previously designed Storm topology, as we cannot reconstruct trajectories directly from disconnected paths. To this end, an additional step is needed to retrieve complete trajectory data and perform a verification task. Figure 10 provides an overall Storm topology, which is an extension of the path query Storm topology. The partially reconstructed trajectories are emitted from *Trajectory Reconstruction Bolt* using *ShuffleGrouping* to the following components:

• **Trajectory Retrieval Bolt.** This bolt retrieves detailed trajectory data from the map-matched Azure Table based on candidate trajectory IDs and their temporal ranges. It then verifies the correctness with the path groups in the querying parameter. After that, the retrieved information is emitted to *Report Bolt* using *FieldsGrouping* mechanism based on path query IDs. To overcome I/O bottleneck, this process is implemented using multiple bolts to retrieve trajectory data from different Azure tables.

• **Report Bolt.** This bolt is fed with qualified trajectories. Once all trajectories with the same path query have arrived, the results are sent to the client via an Azure Redis cache.

6 Experiments & Deployment

In this section, we conduct extensive experiments to evaluate our system. The system is implemented in C#, and deployed on Microsoft Azure. We first describe the details of trajectory dataset used in the experiments. After that, we provide a detailed efficiency study on different parameters on both *index building* and *query processing*. Finally, we show a deployed traffic analysis system based on our path query processing framework.

6.1 Dataset & Settings

In this subsection, we first present the trajectory dataset used in our experiments. After that, we describe the experiment settings, including the configurations of Microsoft Azure and default parameters. Finally, we present a set of experiments to choose a default *max height* and path decomposition method.

6.1.1 Dataset

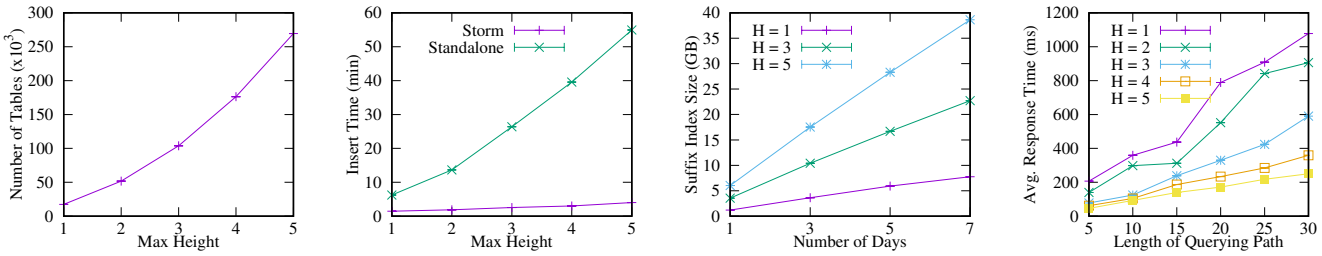
Trajectories. In the experiments, we use real taxi trajectories from Guiyang, a southwest city in China, starting from Sept. 30th, 2016 to Feb. 28th, 2017. The dataset contains 893,001,150 GPS points of 5,425 taxis, whose average sampling interval is about 1 minute. Moreover, to test the scalability of our proposed solution with different numbers of taxis, we extract a copy of historical data, and then randomly sample and copy existing trajectories to simulate usage scenarios with different total numbers of taxis, from 3,000 to 20,000.

Road Networks. We extract the road networks of Guiyang, China from Bing Map. The extracted road networks contain 23,236 vertexes and 29,334 road segments.

6.1.2 Azure Resources

Table 1 summarizes the Azure settings used in our experiments. We use locally redundant storage (LRS) for Table Storage and Blob Storage, where the data is duplicated by three times in the same data center[5]. We also use the Storm component provided by HDinsight, of which the number of data nodes varies from 1 to 15.

5. https://docs.microsoft.com/en-us/azure/storage/storage-redundancy



(a) Suffix Combinations. (b) Indexing Efficiency. (c) Index Sizes. (d) Query Efficiency.

Fig. 11: Max Height Settings.

TABLE 1: Azure Resource Settings

Azure Components	Configurations
Table Storage	Locally Redundant Storage (LRS)
Blob Storage	Locally Redundant Storage (LRS)
Azure Storm	Head Node: A3, 4 cores, 7GB RAM
	Zookeeper: A1, 1 cores, 1.75GB RAM
	Data Node: A3, 4 cores, 7GB RAM

6.1.3 Experiment Parameters

Table 2 shows all parameters in the experiments, where the default settings are highlighted. For the querying efficiency, we randomly select 10 paths as querying paths. For each querying path, we execute path queries for 1,000 times, and calculate the average response time as final results.

TABLE 2: Default Experiment Settings

Component	Parameters	Settings
Indexing	Data Size	3,000, **5,000**, 10,000, 20,000
	Temporal Batch	5, 10, **20**, 30, 40 minutes
	Storm Size	1, 3, **5**, 10 nodes
Querying	Path Length	5, **10**, 15, 20
	Timespan	1, **2**, 3, 4, 5 hours
	Storm Size	1, 3, **5** ,7 ,10, 15 nodes

***Max Height* Settings.** The most important setting in the experiments is the *max height* of suffix tree index. We provide a set of experiments in Figure 11 to demonstrate the effects of different *max heights*.

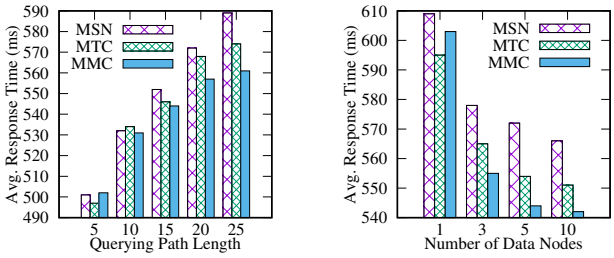
Figure 11a gives the number of suffix combinations generated by trajectories in each batch (i.e., in 20 mins). From the figure, we can see that the number of suffix combinations increases exponentially with the growth of *max height*, because a larger *max height* means more suffix combinations generated by a trajectory. As each suffix combination incurs one insertion to Azure Table, the temporal cost of insertion also grows exponentially, as demonstrated in Figure 11b. It is clear from Figure 11b that using the proposed Storm-based approach, a significant efficiency improvement is achieved, especially with a larger *max height*. Because Storm distribute trajectory data to multiple machines, and can overcome the high I/O overhead problem. For example, we can index a 20-min trajectory batch within four minutes using Storm, while the Standalone approach needs over 50 minutes (which is useless in real-time scenarios).

Moreover, different sizes of suffix data are generated with different *max heights*, as shown in Figure 11c, where a larger H results more duplicated data during the suffix generation process. It is worth noting that it also demonstrates the size of in-memory inverted index for traditional approaches, e.g., [9, 10] with $H = 1$. Thus, it is impossible for in-memory indexing method to scale up when the duration of trajectory data is very long. Our proposed suffix index stores actual trajectory records in Azure Table, and only hold some statistics in memory. Thus, the proposed index can process a very long period of trajectory data.

Query processing with different lengths of querying path is also evaluated with different *max heights*. As shown in Figure 11d, it is obvious that with a longer querying path length, more time is used, as it retrieves more entries from Azure Table. Moreover, with a larger *max height*, the query efficiency is better. Because with a larger *max height*, the index pre-computes more information and generates fewer candidates.

As a result, *max height* is an important trade-off factor in the system that, with a larger *max height*, the system needs more time and more space in the index processing, while the query processing time is saved. To balance the efficiency between the indexing and query processing, for the remaining experiments, *max height* is set as three. However, users can set a different value of *max height* according to different application scenarios.

Path Decomposition Methods. Figure 12a gives the average response time with different lengths of querying path using the three path decomposition methods proposed in this paper, i.e., Minimize Sub-path Number (MSN), Minimize Total Count (MTC) and Minimize Max Count (MMC). All the experiments are based on a Storm cluster with five data nodes. We can observe from the figure that MMC has the best performance, especially, when the length of querying path is larger, as in a distributed computing environment, this method avoids to retrieve suffixes with a lot of entries, which avoids the potential I/O bottleneck. On the other hand, the naive MSN method has the worst performance, as it does not consider the significant differences between suffix entries.



(a) Querying Path Length. (b) Number of Data Nodes.

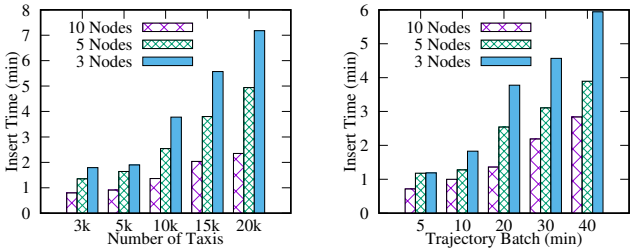
Fig. 12: Path Decomposition Methods.

Figure 12b presents the average response time of three path decomposition methods with different numbers of data nodes in a Storm cluster. We set the querying path length as

20 to make the difference among the path decomposition methods clearer. It shows that, with more data nodes, we can achieve a higher efficiency, because the I/O overhead is distributed among more machines. MMC has the best performance when the number of data nodes is greater than or equal to 3, because it limits the maximum entries retrieved in each node, which avoids the potential I/O bottleneck. For the cluster with one data node, MTC has the best performance, because it reduces the size of data retrieved from the storage. As we deploy our system in a distributed environment for high concurrency queries, MMC is used as our default path decomposition method in the remaining experiments.

6.2 Index Building Efficiency

In this subsection, we evaluate the performance of index building with different settings:



(a) Different Data Size. (b) Different Temporal Batch.

Fig. 13: Index Building Efficiency.

Different Trajectory Data Sizes. Figure 13a gives the suffix tree index building time with different trajectory data sizes, i.e., from 3,000 to 20,000 vehicle trajectories. We test the indexing time for different numbers of nodes in the Storm cluster. From the figure, we have the following observations: 1) the indexing time increases with bigger data sizes, as more trajectories generate more suffix combinations to be inserted in Azure Table. Moreover, the total size of inserted data increases; and 2) with more data nodes, we can see a clear performance boost, as the main bottleneck in this process is the number of I/O accesses to Azure Table. Thus, with more nodes in the Storm cluster, the throughput of system increases significantly and a better efficiency is achieved.

Different Temporal Batch Sizes. Figure 13b gives the index building time with different temporal batch sizes, i.e., from 5 to 40 minutes. We test our system with three different settings of Storm cluster. It is clear that with a longer temporal batch, the processing time increases significantly, as more trajectories being cached in a longer batch. It results in: 1) more road segments are covered, 2) more suffix combinations are generated, and 3) more entries to be inserted into Azure Table. We can also realize that a Storm cluster with more data nodes works better. It is because with more data nodes, Azure Table insertion operations can be executed more effectively in parallel.

6.3 Query Processing Efficiency

In this subsection, the query processing performance is evaluated with different query parameters: 1) trajectory data sizes, 2) trajectory data periods, 3) querying time spans, and 4) length of querying paths. For each experiment, we randomly select 10 querying paths and execute 1,000 times path queries for each path, and then evaluate the average response time. After that, we perform a concurrent query test to demonstrate the scalability of our system, where multiple queries are inserted into the queue at the same time, and we evaluate the average response time of our system to address all queries. Finally, we also show the experiments for the performance of our extension problem, i.e., partial path query processing.

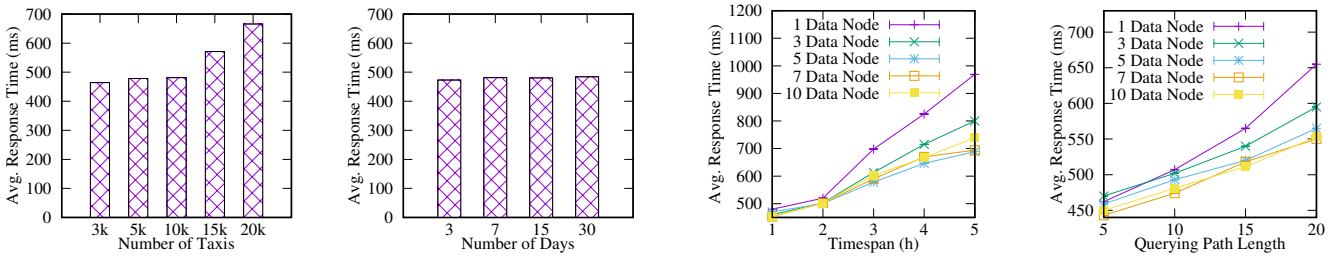
Different Numbers of Trajectories. Figure 14a illustrates the path query processing performance (i.e., response time) with different trajectory sizes from 3,000 to 20,000 vehicles. With more trajectories, the query processing time increases, as more trajectory information is retrieved from the same suffix. It is also very important to notice that the growth of response time is relatively small, comparing to the size of dataset. It is because that trajectory data with the same suffix is stored in the same partition, which can be retrieved efficiently in one batch.

Different Sizes of Historical Data. Figure 14b shows the query response time with different sizes of historical trajectory data, varying from 3 to 30 days. It is obvious from the figure that the querying performance is very consistent with different amount of historical data. It is because the data in Azure Table is partitioned based on their temporal ranges. More historical data only generates more partitions, and Azure storage component has been optimized for the partition-based data access. As a result, we can confirm that our system is scalable to handle trajectory data with very long time period, without degrading the querying performance.

Different Querying Time Spans. Figure 14c gives the path query response time with different time spans from 1 to 5 hours. It is clear from the figure that, with a longer timespan, the response time increases. It is because more data partitions are accessed in Azure Table and more qualified trajectories are returned. It is also interesting to see that, although in general, we can achieve a little better performance with more data nodes, there are very little performance differences when the number of data node is more than 3. The reason behind this is that in the default settings, the length of querying path is 10, which only generates 13 different suffixes, a Storm cluster with 3 nodes is large enough to distribute the workload of suffix record retrieval.

Different Querying Path Lengths. Querying efficiency with different numbers of edges in a querying path is presented in Figure 14d, where the number of edges is set from 5 to 20. It is clear to see that the query processing time increases with more edges in a querying path. It is because that more suffixes are generated, which increases the number of Azure Table accesses. Moreover, we can also observe that with more data nodes, the performance of the query processing is generally better. However, similar to the previous experiment, the performance difference among the Storm clusters with different numbers of data nodes is very limited.

As a result, we can conclude that, if the system is used to answer single path queries from users, the number of data nodes used in the Storm cluster is not a performance bottleneck. In other word, it is a more economic solution to



(a) Different Trajectory Numbers. (b) Different Historical Data Sizes. (c) Different Querying Timespans. (d) Different Query Path Lengths.

Fig. 14: Query Processing Efficiency.

have a small number of data nodes in the Storm cluster for answering path queries individually.

Scalability of Concurrent Queries. In many urban applications, e.g., traffic analysis and path recommendations, a lot of path queries need be answered in batches to complete analysis tasks. For example, 30 path queries with different dates need to be answered if a user wants to analyze the travel time distribution of one path in the morning rush hours through one month. As a result, the scalability of handling concurrent path queries is vital to support complex data mining tasks.

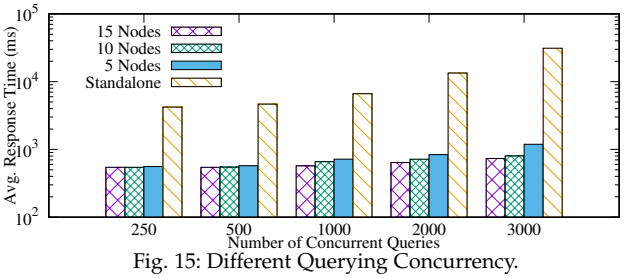
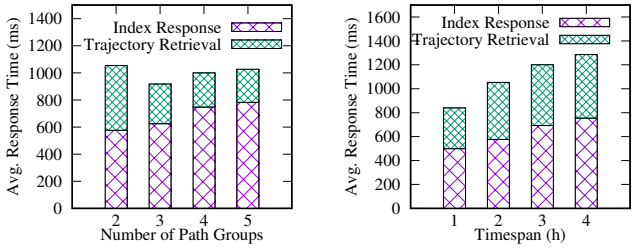


Fig. 15: Different Querying Concurrency.

Figure 15 presents the average response time with different numbers of concurrent queries in the queue, varying from 200 to 3,000. The experiments are done with different numbers of data nodes in the Storm cluster, and the "standalone" method represents a centralized version of the proposed solution. It is clear in the figure that with more concurrent queries in the queue, the average response time increases. Moreover, the Storm-based querying processing framework is much better than Standalone method, and the Storm clusters with more number of data nodes perform better when more concurrent queries are issued in the queue. This is because that, with more data nodes, we can distribute I/O overhead more effectively and achieve a better system throughput.

As a result, we can conclude that if the path query processing system needs to serve concurrent path queries from data analysis/mining applications, more data nodes should be employed in the Storm cluster to ensure the system efficiency.

Partial Path Queries. In this set of experiments, i.e., Figure 16a and 16b, we evaluate the performance of partial path queries. As the system needs to access map-matched results to extract qualified trajectories, the response time in the figure is divided into two parts: 1) index response, which is the time to get data from the suffix index (marked in red); and 2) trajectory retrieval, which is the time to obtain full trajectory information from map-matched tables (marked in green).



(a) Different Path Groups. (b) Different Timespans.

Fig. 16: Partial Query Processing Efficiency.

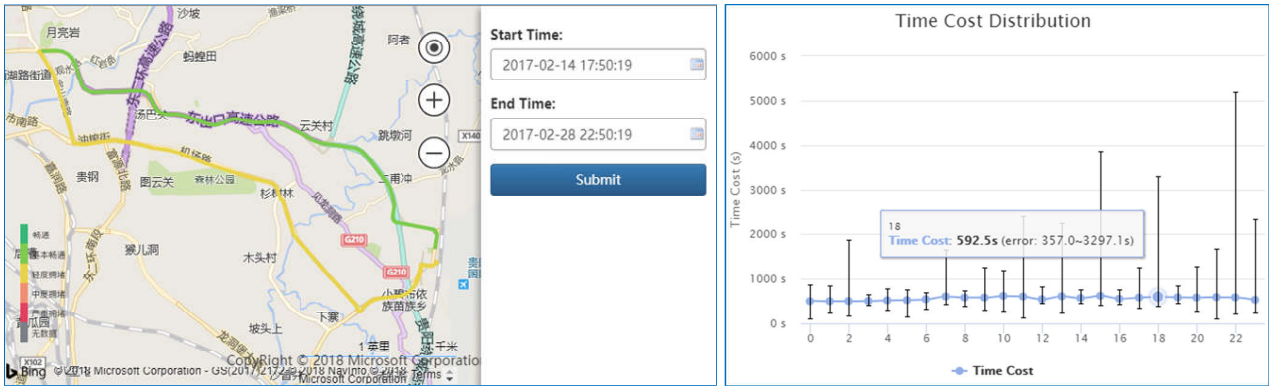
Figure 16a gives the average response time of partial path queries with different numbers of path groups (i.e., the number of disconnected edges in a partial path), from 2 to 5. The figure shows that the time to access the suffix index increases with more path groups, as more suffix entries are accessed to retrieve candidate trajectories. On the other hand, the trajectory retrieval time decreases significantly, because with more path groups in the parameter, fewer trajectories are qualified. As the trajectory retrieval process also needs to access Azure Table, which takes significant response time, we can observe that with more path groups in the parameter, the total response time decreases first, and then increases.

Figure 16b shows the average response time with different time spans in a partial path query processing scenario. We can observe that both index response time and trajectory retrieval time increase with a longer time span. This is because, with a longer time span, more data is returned when accessing the table-based suffix tree index. Also, more qualified trajectories are retrieved from corresponding map-match tables when the time span is longer.

6.4 System Deployment

We deployed a real-time traffic analysis system, entitled *Urban Traffic* [13], on Microsoft Azure, as shown in Figure 17. This system aims to help the Guiyang government to analyze the traffic conditions of two main highways connecting downtown and the airport (highlighted on the map).

At the back-end, the system continuously gets the GPS updates from over 5,000 active taxis in Guiyang City. At the front-end, a user specifies a temporal range for analysis, e.g., the past two weeks, as shown in Figure 17a. As a result, multiple path queries are issueed (one query for each hours), which creates multiple queries for each path. After all qualified trajectories are retrieved, we plot a hourly travel time distribution chart for the user, as shown in Figure 17b.



(a) Main System Interface. (b) Traffic Analysis Panel.

Fig. 17: A Traffic Analysis System in Guiyang Government.

In this chart, the blue line is the average travel time, and the range indicates the travel time variances. In this example, we can see at 22:00, there is a significant peak, which means that some trajectories spent much more time (usually is the indication of traffic jam or accident). This tool is very useful for the government, as they can compare the results from different temporal periods, and generate reports to verify the effectiveness of transportation regulations and operations, such as traffic control or road constructions.

7 Discussion

Although our solution is implemented based on Microsoft Azure, other cloud platforms, e.g. AWS [6], Alibaba Cloud [7], and JD Cloud [8], can also be used, because all of them have already integrated Redis Cache, Kafka Message Queue and Storm Cluster (The three are open-source projects). They also provide substitutions for Azure Table and Azure Blob. For example, Alibaba Cloud provides OSS (Object Storage Service) [9] which can store the suffix tree structure, and Table Store [10] which can store the suffix tree records. In the open-source world, Apache HDFS [11] can replace Azure Blob seamlessly, and Apache HBase [12] could be an alternative of Azure Table. However, as HBase does not support secondary index, and the number of tables in a HBase cluster may be limited, the schema to store suffix tree records using HBase should be carefully redesigned, which is left for our future work.

8 Related Works

We summarize the related works with following two aspects: 1) trajectory query & mining, and 2) parallel spatial computing platforms.

8.1 Trajectory Query & Mining

Trajectories can be extensively used for many urban applications, e.g., urban planning[20], traffic management[21], environment protection[22] and safty monitoring[23–25]. To make use of massive trajectories, many different spatio-temporal indexes have been proposed, such as [26–30]. With road network constraints, new index structures have been proposed to index and query the movements in a spatial network, e.g., [31–34]. However, existing network constrained indexing techniques do not support path queres directly, as they require an additional filtering and refining process, which is inefficient to execute path-based queries and analyses. The closest works on path query processing are [1, 9–11]. [9, 10] process path-based queries with efficient arithmetic operations to verify trajectories and avoid join operations. [1, 11] employ a suffix tree index to speed up join operations. However, these solutions still suffer from three main drawbacks: 1) they need to maintain a trajectory index (e.g., inverted list or suffix tree) in the memory, which is problematic when the size of trajectory data is huge, e.g., millions of trajectories over several years; 2) all of them are implemented based on one single machine, which creates performance bottleneck when a large number of concurrent queries are issued from urban data mining systems; and 3) none of existing works is able to handle real-time trajectory updates, which makes them impossible to process real-time queries/analyses.

Unlike aforementioned approaches, our proposed system introduces a modified suffix tree index with bounded size, and takes advantage of the cloud storage to hold arbitrary size of trajectory data, without worrying about if the size of index can be fit in the memory. We also utilize a parallel streaming framework, i.e., Storm, to overcome the I/O bottleneck in index building and query processing.

8.2 Parallel Spatial Computing Platforms

Parallel computing platforms are not designed originally for spatio-temporal computing. Thus, there are many attempts to extend existing parallel computing platforms to support spatio-temporal data. The first attempt to involve Hadoop in spatial computing is done by Parallel SECONDO [35], which combines Hadoop with SECONDO. Hadoop-GIS [36] utilizes global partition indexes and customized on-demand local spatial indexes to efficient supports multiple types of spatial queries. SpatialHadoop [37] is a comprehensive extension to Hadoop, which has native support for spatial data by modifying the underlying code of Hadoop. [38] and [39] propose parallel algorithms to answer trajectory spatio-temporal range queries using Hadoop framework.

6. https://aws.amazon.com/
7. https://www.alibabacloud.com/
8. https://www.jdcloud.com/
9. https://www.alibabacloud.com/product/oss
10. https://www.alibabacloud.com/product/table-store
11. http://hadoop.apache.org/
12. https://hbase.apache.org/

Most recently, due to the high I/O cost in Hadoop, there are some systems, e.g., SpatialSpark [40], GeoSpark [41], and TrajSpark [42] proposed, trying to support large-scale spatial queries and joins in Spark framework. [15] and [43] build a holistic distributed computing platform for preprocessing and querying trajectory data on the cloud. However, none of above systems supports path queries directly. In our system, special index data structures and parallel computing algorithms are used to enhance the performance of path query processing on the cloud.

9 Conclusion

In this paper, we present a holistic and real-time path query processing system on Microsoft Azure. We modify the original suffix tree index with *max height*, *hourly count* and *table storage* to index large-scale trajectories. The system contains two parts: 1) *back-end processing*, which pre-processes massive raw trajectories and updates table-based suffix index efficiently. In order to support real-time updates, we implement our indexing algorithm based on Storm to overcome the I/O bottleneck; and 2) *service providing*, which answers path queries and partial path queries efficiently with a carefully designed Storm topology. To further improve the efficiency of system, multiple heuristic querying path decomposition methods are proposed.

The experiments, based on the real trajectories of over 5,000 taxis in Guiyang, the capital of Guizhou Province, China, demonstrate the efficiency of our system. We can index a 20-min trajectory batch within less than four minutes, which enables real-time path queries and trajectory analyses. The individual query processing time is less than 400 ms, with a setting of 5 data nodes in a Storm cluster. We also provide some insights on choosing a suitable Storm cluster size based on querying workloads: if the system is used to answer individual path queries, fewer data nodes are needed. Finally, we present a real traffic analysis system based on our path query processing framework, which is currently deployed in Guiyang City.

Acknowledgments

Yu Zheng was supported by the National Natural Science Foundation of China (Grant No. 61672399, No. U1609217). Yanhua Li was supported in part by NSF CRII grant CNS-1657350 and a research grant from DiDi Chuxing Research. Yingcai Wu was supported by NSFC (61761136020, 61502416), Zhejiang Provincial Natural Science Foundation (LR18F020001) and Microsoft Research Asia. Liang Hong is supported by the National Key Research and Development Program of China (Grant No. 2016YFB1000603).

References

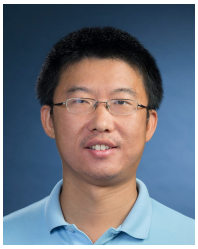
[1] Y. Wang, Y. Zheng, and Y. Xue, "Travel time estimation of a path using sparse trajectories," in *SIGKDD*. ACM, 2014, pp. 25–34.
[2] J. Dai, B. Yang, C. Guo, C. S. Jensen, and J. Hu, "Path cost distribution estimation using trajectory data," *VLDB Endowment*, vol. 10, no. 3, pp. 85–96, 2016.
[3] W. Luo, H. Tan, L. Chen, and L. M. Ni, "Finding time period-based most frequent path in big trajectory data," in *SIGMOD*. ACM, 2013, pp. 713–724.
[4] S. Aljubayrin, B. Yang, C. S. Jensen, and R. Zhang, "Finding non-dominated paths in uncertain road networks," in *SIGSPATIAL*. ACM, 2016, p. 15.
[5] G. Wu, Y. Ding, Y. Li, J. Bao, Y. Zheng, and J. Luo, "Mining spatio-temporal reachable regions over massive trajectory data," in *ICDE*. IEEE, 2017, pp. 1283–1294.
[6] B. Krogh, O. Andersen, and K. Torp, "Trajectories for novel and detailed traffic information," in *SIGSPATIAL*. ACM, 2012, pp. 32–39.
[7] B. Pan, Y. Zheng, D. Wilkie, and C. Shahabi, "Crowd sensing of traffic anomalies based on human mobility and social media," in *SIGSPATIAL*. ACM, 2013, pp. 344–353.
[8] Y. Zheng, L. Capra, O. Wolfson, and H. Yang, "Urban computing: concepts, methodologies, and applications," *TIST*, vol. 5, no. 3, p. 38, 2014.
[9] B. Krogh, N. Pelekis, Y. Theodoridis, and K. Torp, "Path-based queries on trajectory data," in *SIGSPATIAL*. ACM, 2014, pp. 341–350.
[10] B. Krogh, C. S. Jensen, and K. Torp, "Efficient in-memory indexing of network-constrained trajectories," in *SIGSPATIAL*. ACM, 2016, p. 17.
[11] R. Song, W. Sun, B. Zheng, and Y. Zheng, "Press: A novel framework of trajectory compression in road networks," *VLDB Endowment*, vol. 7, no. 9, pp. 661–672, 2014.
[12] R. Li, S. Ruan, J. Bao, Y. Li, Y. Wu, and Y. Zheng, "Querying massive trajectories by path on the cloud," in *SIGSPATIAL*. ACM, 2017.
[13] "Urbantraffic," http://urbantraffic.chinacloudsites.cn.
[14] J. Yuan, Y. Zheng, C. Zhang, X. Xie, and G.-Z. Sun, "An interactive-voting based map matching algorithm," in *MDM*. IEEE, 2010, pp. 43–52.
[15] J. Bao, R. Li, X. Yi, and Y. Zheng, "Managing massive trajectories on the cloud," in *SIGSPATIAL*. ACM, 2016, p. 41.
[16] Y. Zheng, "Trajectory data mining: an overview," *TIST*, vol. 6, no. 3, p. 29, 2015.
[17] R. Li, S. Ruan, J. Bao, and Y. Zheng, "A cloud-based trajectory data management system," in *SIGSPATIAL*. ACM, 2017.
[18] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM (JACM)*, vol. 23, no. 2, pp. 262–272, 1976.
[19] Y. Li, Y. Zheng, S. Ji, W. Wang, Z. Gong *et al.*, "Location selection for ambulance stations: a data-driven approach," in *SIGSPATIAL*. ACM, 2015, p. 85.
[20] J. Bao, T. He, S. Ruan, Y. Li, and Y. Zheng, "Planning bike lanes based on sharing-bikes' trajectories," in *SIGKDD*. ACM, 2017, pp. 1377–1386.
[21] T. He, J. Bao, R. Li, S. Ruan, Y. Li, C. Tian, and Y. Zheng, "Detecting vehicle illegal parking events using sharing bikes trajectories," in *SIGKDD*. ACM, 2018.
[22] Y. Zheng, X. Yi, M. Li, R. Li, Z. Shan, E. Chang, and T. Li, "Forecasting fine-grained air quality based on big data," in *SIGKDD*. ACM, 2015, pp. 2267–2276.
[23] J. Zhang, Y. Zheng, D. Qi, R. Li, X. Yi, and T. Li, "Predicting city-wide crowd flows using deep spatio-temporal residual networks," *Artificial Intelligence*, 2018.
[24] J. Zhang, Y. Zheng, D. Qi, R. Li, and X. Yi, "Dnn-based prediction model for spatio-temporal data," in *SIGSPATIAL*. ACM, 2016, p. 92.
[25] A. Vahedian, X. Zhou, L. Tong, Y. Li, and J. Luo, "Forecasting gathering events through continuous destination prediction on big trajectory data," 2017.
[26] M. F. Mokbel, T. M. Ghanem, and W. G. Aref, "Spatio-temporal access methods," *IEEE Data Eng. Bull.*, vol. 26, no. 2, pp. 40–49, 2003.
[27] Y. Li, C.-Y. Chow, K. Deng, M. Yuan, J. Zeng, J.-D. Zhang, Q. Yang, and Z.-L. Zhang, "Sampling big trajectory data," in *CIKM*. ACM, 2015, pp. 941–950.
[28] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire, "A gpu-based index to support interactive spatio-temporal queries over historical data," in *ICDE*. IEEE, 2016, pp. 1086–1097.
[29] N. Ferreira, J. Poco, H. T. Vo, J. Freire, and C. T. Silva, "Visual exploration of big spatio-temporal urban data: A study of new york city taxi trips," *TVCG*, vol. 19, no. 12, pp. 2149–2158, 2013.
[30] Y. Ding, Y. Li, X. Zhou, Z. Huang, S. You, and J. Luo, "Sampling big trajectory data for traversal trajectory aggregate query," *IEEE Transactions on Big Data*, 2018.
[31] V. T. De Almeida and R. H. Güting, "Indexing the trajectories of moving objects in networks," *GeoInformatica*, vol. 9, no. 1, pp. 33–60, 2005.
[32] E. Frentzos, "Indexing objects moving on fixed networks," in

SSTD. Springer, 2003, pp. 289–305.
[33] D. Pfoser and C. S. Jensen, "Indexing of network constrained moving objects," in *SIGSPATIAL*. ACM, 2003, pp. 25–32.
[34] I. Sandu Popa, K. Zeitouni, V. Oria, D. Barth, and S. Vial, "Indexing in-network trajectory flows," *VLDBJ*, vol. 20, no. 5, pp. 643–669, 2011.
[35] J. Lu and R. H. Güting, "Parallel secondo: boosting database engines with hadoop," in *ICPADS*. IEEE, 2012, pp. 738–743.
[36] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," *VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.
[37] A. Eldawy and M. F. Mokbel, "A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data," *VLDB Endowment*, vol. 6, no. 12, pp. 1230–1233, 2013.
[38] Q. Ma, B. Yang, W. Qian, and A. Zhou, "Query processing of massive trajectory data based on mapreduce," in *Proceedings of the first international workshop on Cloud data management*. ACM, 2009, pp. 9–16.
[39] L. Alarabi, "St-hadoop: A mapreduce framework for big spatio-temporal data," in *SIGMOD*. ACM, 2017, pp. 40–42.
[40] S. You, J. Zhang, and L. Gruenwald, "Spatial join query processing in cloud: Analyzing design choices and performance comparisons," in *ICPPW*. IEEE, 2015, pp. 90–97.
[41] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *SIGSPATIAL*. ACM, 2015, p. 70.
[42] Z. Zhang, C. Jin, J. Mao, X. Yang, and A. Zhou, "Trajspark: A scalable and efficient in-memory management system for big trajectory data," in *APWeb/WAIM*. Springer, 2017, pp. 11–26.
[43] S. Ruan, R. Li, J. Bao, T. He, and Y. Zheng, "Cloudtp: A cloud-based flexible trajectory preprocessing framework," 2018.

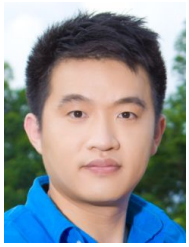
Ruiyuan Li is a Ph.D. student at the School of Computer Science and Technology, Xidian University, China. He received his B.E. degree and M.S. degree from Wuhan University, Hubei, China in 2013 and 2016, respectively. His research focuses on Urban Computing, Spatio-temporal Data Management, and Distributed Computing. He has interned in Urban Computing Group, Microsoft Research Asia from 2014 to 2017, and is now an intern student in JD Intelligent City Research and JD Urban Computing Business Unit under the advisory of Prof. Yu Zheng and Dr. Jie Bao.

Sijie Ruan is a first-year Ph.D. student in the School of Computer Science and Technology, Xidian University. He received his B.E. degree from Xidian University in 2017. His research interests include urban computing, spatio-temporal data mining, and distributed systems. He was an intern in MSR Asia from 2016 to 2017. He is now a research intern in JD Intelligent City Research and JD Urban Computing Business Unit, under the supervision of Prof. Yu Zheng and Dr. Jie Bao.

Jie Bao got his Ph.D degree in Computer Science from University of Minnesota at Twin Cities in 2014. He worked as a researcher in Urban Computing Group at MSR Asia from 2014 to 2017. He currently leads the Data Platform Division in JD Urban Computing Business Unit. His research interests include: Spatio-temporal Data Management/Mining, Urban Computing, and Location-based Services.



Yanhua Li received two Ph.D. degrees in electrical engineering from Beijing University of Posts and Telecommunications, Beijing in China in 2009 and in computer science from University of Minnesota at Twin Cities in 2013, respectively. He has worked as a researcher in HUAWEI Noah's Ark LAB at Hong Kong from Aug 2013 to Dec 2014, and has interned in Bell Labs in New Jersey, Microsoft Research Asia, and HUAWEI research labs of America from 2011 to 2013. He is currently an Assistant Professor in the Department of Computer Science at Worcester Polytechnic Institute (WPI) in Worcester, MA. His research interests are big data analytics and urban computing in many contexts, including urban network data analytics and management, urban planning and optimization.



Yingcai Wu is a tenure-track assistant professor at the State Key Lab of CAD & CG, Zhejiang University, Hangzhou, China. He has been selected by China's 1000-Talents Program for young scholars in 2016. His research interests lie broadly in visual analytics and visualization. He received his Ph.D. degree in Computer Science from The Hong Kong University of Science and Technology (HKUST), Hong Kong in 2009 and obtained his B.Eng. degree in Computer Science and Technology from South China University of Technology, Guangzhou, China in 2004. Prior to his current position, he was a researcher in Microsoft Research from May 2012 to January 2015. He was a postdoctoral researcher at the Visualization research group in HKUST from January to May 2010, and at the Visualization and interface Design Innovation (VIDi) research group in the University of California, Davis from June 2010 to March 2012.

Liang Hong received the BS and PhD degrees in computer science from the Huazhong University of Science and Technology (HUST) in 2003 and 2009, respectively. Now, he is an associate professor in the School of Information Management at Wuhan University. His research interests include knowledge graph, spatio-temporal data management, and social networks. He is a member of the IEEE.

Yu Zheng is a Vice President and Chief Data Scientist at JD Finance Group, passionate about using big data and AI technology to tackle urban challenges. His research interests include big data analytics, spatio-temporal data mining, machine learning, and artificial intelligence. He also leads the JD Urban Computing Business Unit as the president and serves as the director of the JD Intelligent City Research. Before Joining JD, he was a senior research manager at Microsoft Research. Zheng is also a Chair Professor at Shanghai Jiao Tong University, an Adjunct Professor at Hong Kong University of Science and Technology.