Querying Massive Trajectories by Path on the Cloud*

Ruiyuan Li[1,2] Sijie Ruan[1,2] Jie Bao[2] Yanhua Li[3] Yingcai Wu[4] Yu Zheng[1,2,5†]
[1]School of Computer Science and Technology, Xidian University, China
[2]Urban Computing Group, Microsoft Research, Beijing, China
[3]Worcester Polytechnic Institute, Worcester, US
[4]College of Computer Science, Zhejiang Univeristy, Hangzhou, China
[5]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
{v-ruiyli, jiebao, yuzheng}@microsoft.com sjruan94@gmail.com yli15@wpi.edu ycwu@zju.edu.cn

ABSTRACT

A path query aims to find the trajectories that pass a given sequence of connected road segments within a time period. It is very useful in many urban applications, e.g., 1) traffic modeling, 2) frequent path mining, and 3) traffic anomaly detection. Existing solutions for path query 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. In this paper, we design and implement a cloud-based path query processing framework based on Microsoft Azure. We modify the suffix tree structure to index the trajectories using Azure Table. The proposed system consists of two main parts: 1) *backend processing*, which performs the pre-processing and suffix index building with 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 the I/O bottleneck. We evaluate the performance of our proposed system based on a real taxi dataset from Guiyang, China.

CCS CONCEPTS

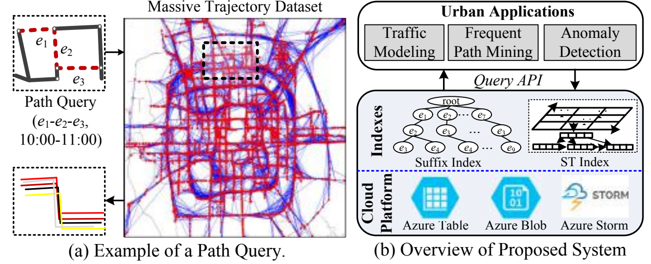
• **Information systems → Spatial databases and GIS**;

KEYWORDS

Trajectory Query Processing, Spatio-temporal Data Management

1 INTRODUCTION

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. As shown in Figure 1a, a user retrieves the trajectories that have passed edges ($e_1 \rightarrow e_2 \rightarrow e_3$), i.e., the red dotted lines during the 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, e.g., 1) traffic speed modeling [1, 2], where the extracted trajectories can be used to estimate the travel



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

Figure 1: Motivating Examples.

time/speed of a given path; 2) path mining [3, 4], where the trajectories can be used for route recommendations; and 3) traffic anomaly detection [5, 6], where we can find anomalous vehicles.

The existing solutions for path query [1, 7, 8] suffer from three main drawbacks: 1) they need to maintain an index (e.g., inverted list or suffix tree) in the memory, which is unrealistic when trajectory data is huge; 2) they are implemented based on a single machine, which incurs a performance bottleneck for answering a large number of concurrent path queries from a data mining task; 3) they do not efficiently handle large scale trajectory updates, which prevents them from providing real-time path query/analysis.

In this paper, we build a cloud-based path query processing system [9] on Microsoft Azure. 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. The distributed streaming computing platform, i.e., Storm, is extensively used to overcome the I/O bottleneck.

Our main idea is to modify the traditional suffix tree index: 1) we set a *max height* to limit the size of the index; 2) we keep an *hourly count* on each suffix record to record 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 (i.e., *max height*) index, we propose a heuristic method to decompose query paths, retrieve them from Azure Table, and efficiently reconstruct the results. Our main contributions are as follows:

• We develop the *table-based suffix tree index*, with *max height*, *hourly count*, and *table storage* to overcome challenges from indexing and querying the massive historical trajectory data.

• We propose an efficient indexing algorithm based on Azure Storm to distribute the system IO overhead and to handle real-time trajectory updates.

• We propose a Storm topology to answer the path queries both individually and concurrently. To further improve performance, a heuristic path decomposition method is proposed.

• Extensive experiments are conducted based on the real taxi trajectories of Guiyang to demonstrate the efficiency of our system.

*The work was supported by the National Natural Science Foundation of China (Grant No. 61672399, No. U1609217, and No. U1401258), NSF CRII grant CNS-1657350 and a research grant from Pitney Bowes Inc.

†Yu Zheng is the correspondence author of this paper.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGSPATIAL'17, November 7–10, 2017, Los Angeles Area, CA, USA

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5490-5/17/11.
https://doi.org/10.1145/3139958.3139996

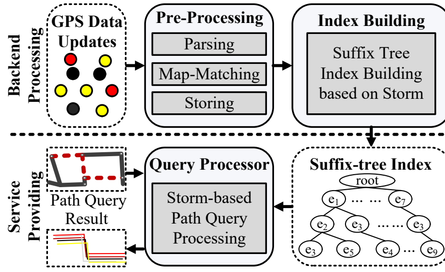


Figure 2: System Overview.

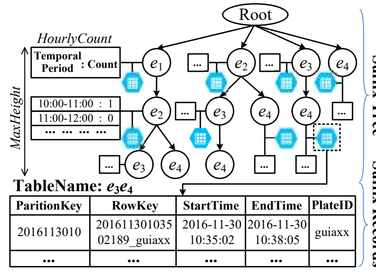


Figure 3: Table-Based Suffix Tree.

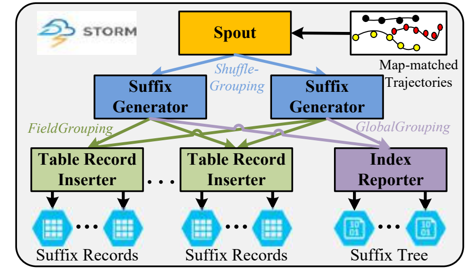


Figure 4: Storm-Based Indexing Topology.

2 PRELIMINARY

2.1 Basic Concepts

Definition 1. *GPS Trajectory.* A GPS trajectory is a time-ordered sequence of GPS points: $tr = \{(p_1, t_1) \rightarrow (p_2, t_2) \rightarrow ... \rightarrow (p_n, t_n)\}$.

Definition 2. *Road Network.* A road network can be viewed as a directed graph $G = (V, E)$, where E is the set of road segments, and V represents the intersections.

Definition 3. *Map-Matched Trajectory.* A map-matched trajectory is a list of road segments and timestamps mapped from GPS points, i.e., $Tr = \{(e_1, t_1) \rightarrow (e_2, t_2) \rightarrow ... \rightarrow (e_n, t_n)\}$.

Definition 4. *Path.* A path is a series of connected road segments $P = \{e_i \rightarrow e_j \rightarrow ... \rightarrow e_k\}$, where the order of the edges indicates the travel sequence. The consecutive road segments in a path should be connected on the road network G.

2.2 Azure Preliminary

2.2.1 Azure Storage. Azure storage is a massive scalable cloud storage solution [1] with many different components:

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

Azure Table. Azure Table is the NoSQL database in Azure. Table storage is a key/attribute store with a schema-less design. Each table entity is identified by: `PartitionKey` and `RowKey`. Table entities with the same `PartitionKey` are stored in the same physical location. Azure Table is very efficient in answering the range queries of `RowKey` within the same `PartitionKey`. Thus, Azure Table is used extensively to store trajectories (more details in [10]).

2.2.2 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, which are essential to many urban applications. There are two types of components in a Storm system: 1) *Spout*, which continuously reads and distributes updates/requests; and 2) *Bolt*, which is the processing unit. *Spouts* and *Bolts* with different functions are connected as a direct acyclic graph (DAG), forming the *Storm Topology*.

2.3 Problem Definition

Path Query. The path query can be formalized as follows: given a path $P = \{e_i, e_j, ..., e_k\}$, a temporal range $[T_s, T_e]$, and a map-matched trajectory dataset $T = \{Tr_1, Tr_2, ..., Tr_n\}$, we want to find all the sub-trajectories of Tr_i in T, where Tr_i passed 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.

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

2.4 System Overview

Figure 2 gives an overview of the proposed system, which is comprised of two components:

Backend Processing. This component receives GPS updates and builds the index, illustrated as the upper part of Figure 2, with two modules: 1) *Pre-Processing*, which gets the raw GPS updates, and performs parsing, map-matching, and storing tasks (Detailed in [10]); and 2) *Index Building*, which builds the suffix tree index to speed up path query processing (Detailed in Section 3).

Service Providing. This component answers path queries, illustrated as the bottom part of Figure 2. The main module is the *Query Processor*, which takes advantage of the suffix tree index and employs the Storm parallel computing platform to answer queries (Detailed in Section 4).

3 INDEX BUILDING

The traditional suffix tree is originally used to index strings [11] for string suffix searching. The trajectory data management system [7] utilizes the suffix tree to efficiently answer path queries, regarding each edge ID as a character. As the traditional suffix tree takes a lot of space and does not support the temporal predicate, it only holds the most frequent or recent trajectories.

To address the aforementioned problems, we propose a table-based suffix tree, as shown in Figure 3, which consists of two components: 1) suffix tree index, which includes the tree structure and a set of statistics stored in Azure Blob and loaded in the memory during the query processing; and 2) suffix records, which store the trajectory data based on suffixes in Azure Table. There are three changes to the traditional suffix tree:

•*Max Height.* We set a max height H to limit the total size of the suffix tree. This way, we can guarantee that the index can fit in the memory regardless of the size of the trajectory data, as it is bound by the number of sub-paths with H edges.

• *Hourly Count.* In each node of the suffix tree, we maintain a hash table, where we store the average number of trajectories passing the corresponding sequence of the edges each hour.

• *Table Storage.* Each node also keeps a pointer to an Azure table, where the actual trajectory data is stored. The name of the table is the sub-path ID (table e_3e_4 in the example). The `PartitionKey` of the Azure Table is the temporal range, e.g., by hour in the figure, and the `RowKey` is the timestamp with the trajectory ID. In this way,

trajectories passing the same path at the same time are stored in the same Azure Table partition for more efficient access.

It is worth noting that we store an extra and re-organized copy of the trajectory dataset, which is different from the traditional indexing methods. Because 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 about 10 USD per 1TB/month [2].

There are three main steps in constructing the table-based suffix tree: 1) *Suffix Generation*, which truncates map-matched trajectories into sub-trajectories whose lengths are not greater than H. E.g, if $H = 2$, the map-matched trajectory $Tr : e_1 \rightarrow e_2 \rightarrow e_3$ will be broken into 5 sub-trajectories: $e_1, e_2, e_3, e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$. 2) *Index Updating*, which groups the suffixes and updates the *hourly count* at each node. 3) *Record Insertion*, which inserts the grouped sub-trajectories into Azure Table, where the sub-trajectories with the same suffix are inserted into the same table, and the sub-trajectories within the same time period are inserted into the same partition.

There are a large number of sub-trajectories generated and numerous records to be inserted into many tables, which may cause CPU and I/O bottlenecks in a single machine. To this end, we build the index based on Storm, as shown in Figure 4, which consists of four types of components: 1) *Spout*, which distributes the map-matched trajectories to different Suffix Generator Bolts, using the `ShuffleGrouping` method to achieve load balance. 2) *Suffix Generator Bolt*, which breaks trajectories into suffixes with the maximum length of H. The generated suffixes are then distributed to two locations: Table Record Inserter Bolts, using the `FieldGrouping` method, i.e., the same suffixes are emitted to the same bolt; and Index Reporter Bolt, using the `GlobalGrouping` method, i.e., all suffixes are emitted to a single bolt. 3) *Table Record Inserter Bolt*, which inserts the sub-trajectories into Azure Table in batches. 4) *Index Reporter Bolt*, which aggregates all suffixes and updates the suffix tree: adding new branches, if a new suffix is generated; and updating the statistics (i.e., hourly count) on each node. After the index is updated, it is stored in Blob.

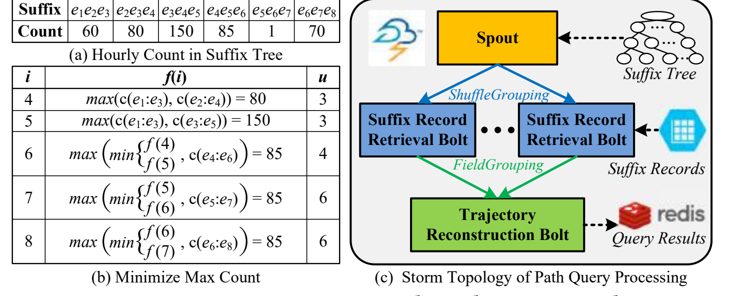
4 QUERY PROCESSOR

As the *table-based suffix tree* employs a *maximum height* H, the qualified trajectories cannot be directly retrieved from the index if the length of the query path is longer than H. Therefore, we propose a new query process and implement it in Azure Storm to improve its efficiency. The proposed query process contains three main steps: 1) *Querying Path Decomposition*, 2) *Suffix Record Retrieval*, and 3) *Sub-Trajectory Reconstruction*.

Step 1. Query Path Decomposition. This step breaks the query path P into several sub-paths with a maximum length of H, where each sub-path can be mapped as a table. To effectively take advantage of the indexed suffix records in Azure Table, the following three requirements should be satisfied:

1) $\forall p_i \in P, |p_i| = H$, i.e., the lengths of the decomposed sub-paths should be equal to the *maximum height* H, as it returns the fewest qualified candidates from the storage.

2) $P = \bigcup_{i=1}^{l} p_i$, which means all edges e_i in P should be covered in the union set of decomposed sub-paths, as fewer candidates are returned.



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

(b) Minimize Max Count

(c) Storm Topology of Path Query Processing

Figure 5: Min Max Count Example and Storm Topology.

3) $\forall p_i, p_{i+1} \in P, p_i \bigcap p_{i+1} \neq \emptyset$, i.e., every pair of consecutive sub-paths has at least one overlapped edge. Otherwise, the continuity and order cannot be guaranteed.

There are multiple ways to decompose a query path. An intuitive method uses a sliding window, with a window size of H and a step size of $H - 1$, to decompose the query path. The main intuition is to minimize the decomposed sub-path count, which essentially minimizes the total number of sub-trajectory retrievals to the table-based suffix tree.

However, the query performance depends on the slowest retrieval in a parallel environment. Therefore, we develop a dynamic programming algorithm to minimize the max count of retrieved sub-trajectories in all decomposed sub-paths. The state transfer equation is:

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

where $f(i)$ denotes the minimum max count when a path ends with the edge e_i, and $c(e_j : e_i)$ denotes the *hourly count* for the corresponding path $e_j \rightarrow e_{j+1} \rightarrow \dots \rightarrow e_i$. In each step, we keep an array $\boldsymbol{u}$ to record the decomposition slot.

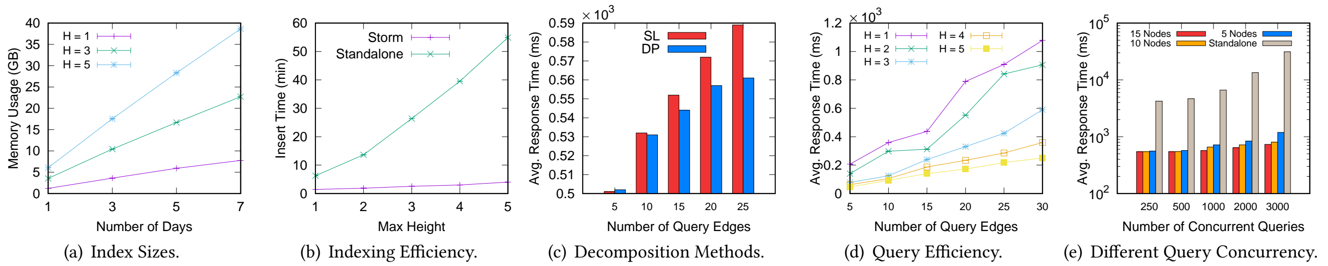
Figure 5 gives an example of decomposing a query path $P = \{e_1 \rightarrow \dots \rightarrow e_8\}$ with $H = 3$, where the *hourly count* of each sub-path is in Figure 5a and the decomposing steps are presented in Figure 5b. As a result, the path P is decomposed to four sub-paths: $p_1 = \{e_1 : e_3\}, p_2 = \{e_2 : e_4\}, p_3 = \{e_4 : e_6\}$, and $p_4 = \{e_6 : e_8\}$.

Step 2. Suffix Record Retrieval. In this step, the system retrieves the sub-trajectories based on the decomposed plan from Azure Table. We regard the decomposed path as the table name and temporal range as the key range. As a result, each query to the *table-based suffix index* returns a set of suffix records, which consists of its ID, and a pair of start/end times of the corresponding sub-path.

Step 3. Sub-Trajectory Reconstruction. In this step, we reconstruct the qualified trajectories by joining the sub-trajectories. The suffix record of the sub-trajectories from the first sub-path is our candidate set. For each sub-trajectory in the candidate set, if it does not appear in the next sub-path, the trajectory is discarded. Otherwise, we check if these sub-trajectories have the correct overlapped time between their start and end timestamps.

To overcome the I/O bottleneck and efficiently support large-scale concurrent path queries, we implement the query processing component using Storm, as shown in Figure 5c, with three main modules: 1) *Spout*, which reads the query parameters from users, assigns IDs to different queries, decomposes the query paths and emits the decomposed sub-paths with their query IDs to *Suffix*

[2] https://azure.microsoft.com/en-us/pricing/details/storage/



(a) Index Sizes. (b) Indexing Efficiency. (c) Decomposition Methods. (d) Query Efficiency. (e) Different Query Concurrency.

Figure 6: Experiment Results.

Record Retrieval Bolts, using the ShuffleGrouping mechanism to balance the workloads; 2) *Suffix Record Retrieval Bolt*, which gets the decomposed sub-paths and retrieves the sub-trajectories from Azure Table. The retrieved sub-trajectories are then emitted to the *Trajectory Reconstruction Bolt*, using the FieldsGrouping mechanism based on their IDs (i.e., the sub-trajectories from the same query are passed to the same bolt); 3) *Trajectory Reconstruction Bolt*, which receives the sub-trajectories with the same query ID, and reconstructs the overlapped sub-trajectories for each query. Finally, the qualified trajectories are written as results to the Redis server.

5 EXPERIMENTS

Dataset & Settings. We use one month of GPS trajectories from around 5,000 daily active taxis in Guiyang (the capital of Guizhou Province, China), whose average sampling interval is about 1 minute. The raw trajectories and map-matched trajectories are about 7GB and 61GB in disk space, respectively. For Azure settings, we use locally redundant storage (LRS) for the Table and Blob Storage. We also use the Storm component, whose default number of data nodes used in experiments is 5.

Index Performance. Figure 6a shows the index sizes of different max heights H growing with the increasing number of days, where a larger H results in more duplicated data during the suffix generation process. Figure 6b shows that the insert time by processing 20-min trajectories varies with different H. It is clear that with a larger H, the insert time grows exponentially, because the number of suffix combinations increase exponentially with the growth of H, and each suffix combination incurs one insertion to the Azure Table. As the Storm distributes the suffix combinations to multiple machines, it achieves significant efficiency improvement, especially with a larger H. For example, we can index the 20-min trajectories within four minutes using Storm, while the standalone needs over 50 minutes (which is useless in real-time scenarios).

Query Efficiency. Figure 6c gives the average response time with different edge sizes using two path decomposition methods, i.e., Sliding Window method (SW), and Dynamic Programming method (DP). It is clear that DP performs better, especially with more edges. As in the distributed environment, DP avoids retrieving the suffixes with many entries, while SW does not consider the significant size differences in each suffix node. As a result, DP is used as our default decomposition method.

Query processing with different numbers of query paths is also evaluated with different H. As shown in Figure 6d, it is obvious that with more edges, 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 H, the index pre-computes more information and generates fewer candidates.

Scalability of Concurrent Queries. The scalability of handling concurrent path queries is vital to many complex data mining tasks. Figure 6e presents the average response time with different numbers of concurrent queries, using different numbers of data nodes in Storm. It is clear that with more concurrent queries, the average response time increases. Moreover, the Storm with more data nodes performs better, as with more data nodes we can distribute the I/O access more effectively and achieve a better system throughput.

Result. Choosing the suitable *max height* H is a trade-off: if H is small, it favors index building, as fewer suffixes are generated. However, it hurts the efficiency in query processing, as more join operations and data accesses are introduced. If the path query processing system needs to serve concurrent path queries for data analysis/mining applications, more data nodes should be employed in the Storm to ensure system efficiency.

6 CONCLUSION

We present a holistic path query processing system on Microsoft Azure. We modify the suffix tree, with *max height*, *hourly count* and *table storage* for indexing large scale trajectories. A heuristic path decomposition method is developed to further improve performance. Experiments on a real dataset verify the efficiency of our system. For example, when $H = 3$, we can index 20-min trajectories in less than 4 minutes, which enables real-time path querying and analysis. The individual query processing time is less than 400 ms.

REFERENCES

[1] Yilun Wang, Yu Zheng, and Yexiang Xue. Travel time estimation of a path using sparse trajectories. In *SIGKDD*, pages 25–34, 2014.

[2] Jian Dai, Bin Yang, Chenjuan Guo, Christian S Jensen, and Jilin Hu. Path cost distribution estimation using trajectory data. *VLDB*, 10(3):85–96, 2016.

[3] Saad Aljubayrin, Bin Yang, Christian S Jensen, and Rui Zhang. Finding non-dominated paths in uncertain road networks. In *SIGSPATIAL*, page 15, 2016.

[4] Guojun Wu, Yichen Ding, Yanhua Li, Jie Bao, Yu Zheng, and Jun Luo. Mining spatio-temporal reachable regions over massive trajectory data. In *ICDE*, 2017.

[5] Bei Pan, Yu Zheng, David Wilkie, and Cyrus Shahabi. Crowd sensing of traffic anomalies based on human mobility and social media. In *SIGSPATIAL*, pages 344–353. ACM, 2013.

[6] Junbo Zhang, Yu Zheng, Dekang Qi, Ruiyuan Li, and Xiuwen Yi. Dnn-based prediction model for spatio-temporal data. In *SIGSPATIAL*, page 92. ACM, 2016.

[7] Renchu Song, Weiwei Sun, Baihua Zheng, and Yu Zheng. Press: A novel framework of trajectory compression in road networks. *VLDB*, 7(9):661–672, 2014.

[8] Benjamin Krogh, Nikos Pelekis, Yannis Theodoridis, and Kristian Torp. Path-based queries on trajectory data. In *SIGSPATIAL*, pages 341–350. ACM, 2014.

[9] Ruiyuan Li, Sijie Ruan, Jie Bao, and Yu Zheng. A cloud-based trajectory data management system. In *SIGSPATIAL*. ACM, 2017.

[10] Jie Bao, Ruiyuan Li, Xiuwen Yi, and Yu Zheng. Managing Massive Trajectories on the Cloud. In *ACM SIGSPATIAL*, 2016.

[11] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.