Managing Massive Trajectories on the Cloud

Jie Bao[1] Ruiyuan Li[1,2] Xiuwen Yi[4,1] Yu Zheng[1,2,3] *

[1]*Microsoft Research, Beijing, China*
[2]*School of Computer Science and Technology, Xidian University, China*
[3]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China*
[4]*School of Information Science and Technology, Southwest Jiaotong University, Chengdu, China*
{jiebao, v-ruiyli, v-xiuyi, yuzheng}@microsoft.com

ABSTRACT

With advances in location-acquisition techniques, such as GPS-embedded phones, an enormous volume of trajectory data is generated, by people, vehicles, and animals. This trajectory data is one of the most important data sources in many urban computing applications, e.g., traffic modeling, user profiling analysis, air quality inference, and resource allocation.

To utilize large scale trajectory data efficiently and effectively, cloud computing platforms, e.g., Microsoft Azure, are the most convenient and economic way. However, traditional cloud computing platforms are not designed to deal with spatio-temporal data, such as trajectories. To this end, we design and implement a holistic cloud-based trajectory data management system on Microsoft Azure to bridge the gap between trajectory data and urban applications. Our system can efficiently store, index, and query large trajectory data with three functions: 1) trajectory ID-temporal query, 2) trajectory spatio-temporal query, and 3) trajectory map-matching. The efficiency of the system is tested and tuned based on real-time trajectory data feeds. The system is currently used in many internal urban applications, as we will illustrate using case studies.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

Keywords

Spatio-temporal Data Management, Trajectory Data Management, Cloud Computing, Microsoft Azure.

1. INTRODUCTION

With advances in location-acquisition techniques,such as GPS-embedded phones, an enormous volume of trajectory data is generated, by people, vehicles, and animals. As demonstrated in Figure 1, processing and managing this massive trajectory data serves as the foundation of many urban computing applications [1]. Many of the urban data mining/machine learning models [2] are based on the trajectory data, e.g.,: 1) air quality inference and predication [3, 4], where the model executes a large amount of spatio-temporal range queries (i.e., looking for the partial trajectories that fall in the given spatial and temporal range); and 2) traffic modeling [5, 6], where the application requires the availability of the real-time map-matched trajectory information (i.e., mapping the raw trajectory points onto the corresponding road network segment).

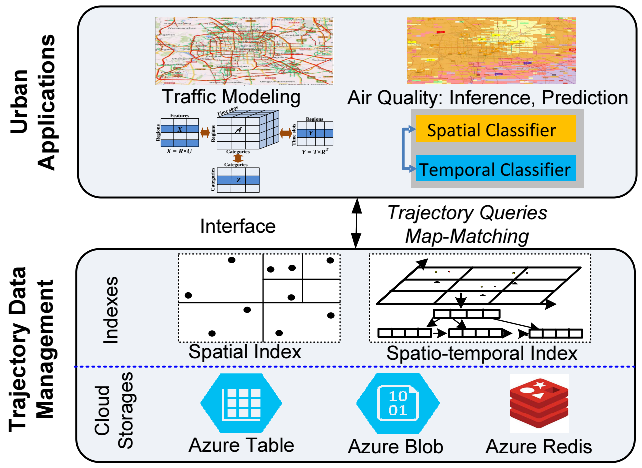


Figure 1: Motivation Scenarios.

Cloud computing platforms, e.g., Microsoft Azure, offer a faster, more reliable, and economical option for users and companies to store and compute large-scale data, by providing large scale storage components, like Azure Blob and Table, and many parallel computing techniques, such as Hadoop, Spark and Storm. However, conventional cloud computing platforms are not optimized to deal with spatio-temporal data sets, like trajectories. In other words, it cannot process spatio-temporal queries over the trajectories efficiently.

In recent years, there have been some attempts to improve spatial query efficiency in the cloud/parallel computing environments,e.g., Hadoop-GIS [7], SpatialHadoop [8], SpatialSpark [9], and GeoSpark [10]. However, all of the aforementioned techniques only work efficiently on managing and querying static geo-spatial data, e.g., POIs and Road networks. To the best of our knowledge, there is no a holistic system on adapting spatio-temporal datasets (most importantly, trajectory data) in the cloud/parallel computing platforms to support real time large scale trajectory data mining and querying, and hence to offer help for the urban computing applications.

*The work was supported by the National Natural Science Foundation of China (Grant No. 61672399 and No. U1401258), the China National Basic Research Program (973 Program, No. 2015CB352400), and NSFC Grant No. 61572488.

SIGSPATIAL'16, October 31-November 03, 2016, Burlingame, CA, USA
© 2016 ACM. ISBN 978-1-4503-4589-7/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2996913.2996916

To this end, we develop a holistic cloud-based trajectory data management system to bridge the gap between cloud computing platforms and massive trajectory datasets. We identify three important tasks in utilizing large trajectory datasets in urban applications: 1) *ID-temporal query*, which retrieves partial trajectories within a given temporal period and a trajectory ID, e.g., finding the route of a taxi traveled from 10:00 AM to 10:30 AM; 2) *spatio-temporal range query*, which retrieves trajectories within a given spatio-temporal period, e.g., finding all partial trajectories that pass through a downtown area from 10:00 AM to 11:00 AM; and 3) *map-matching*, which maps raw GPS points in a trajectory onto its corresponding road segment. The objective of our system is to provide an efficient solution to support full-fledged urban data mining applications: both large scale mining (in the model building phase) and real-time service providing (in the online inference phase).

To support these important functionalities in the cloud system,i.e., Microsoft Azure, in a more efficient way, we optimize the storage schema based on different functionalities, and use parallel computing environments, e.g., Storm. As a result, our system can accept high volume trajectory updates and perform services in real-time. The proposed system includes three main modules: 1) *trajectory storage module*, which stores trajectory data based on its trajectory ID in Azure Table, as well as an extra copy on a Redis server; 2) *trajectory spatio-temporal indexing module*, which creates a copy of trajectory data based on their spatio-temporal properties to speed up trajectory spatio-temporal query processing; and 3) *trajectory map-matching module*, which utilizes Apache Storm, a parallel streaming platform, to map the massive amount of GPS points to the corresponding road segment in real-time. The contributions of the paper are summarized as follows:

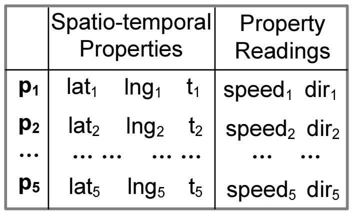
- We provide the first attempt to adapt a cloud computing platform to work efficiently with large amounts of trajectory data both in cloud storage components and computing components.
- The trajectory data in the system is stored in Azure Table with a properly designed storage schema and indexed to answer ID-temporal queries. We also maintain another copy of the trajectory data in the system with a spatio-temporal index to support spatio-temporal queries.
- With the massive trajectory stored in the Azure storage component, we leverage the Storm platform to perform the map-matching service. In this way, we are able to provide the service in a real-time manner.
- We evaluate our system design with the real taxi trajectories updated continuously from Guiyang City, China. The experiment results provide some lessons and insights on how to tune the parameters in Azure services.
- We also demonstrate the capability of our cloud-based trajectory data management system via three real case study systems: 1) real-time taxi data management, 2) real-time traffic modeling, and 3) trajectory-based resource allocation.

Section 2 introduces preliminary details about trajectory data, the basics of Microsoft Azure, and provides a full picture of our system. The three main components: 1) trajectory data store, 2) trajectory spatio-temporal indexing, and 3) trajectory map-matching, are described in Section 3, Section 4 and Section 5, respectively. Our experiments are explained in Section 6. Three case studies are outlined in Section 7. The related works are summarized in Section 8. Finally, Section 9 concludes the paper.

2. PRELIMINARY

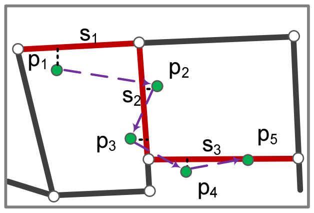
In this section, we first provide the definition of trajectory data, then provide an introduction of the cloud computing components we use in the system from Microsoft Azure. Finally, an overview of our system is presented.

2.1 Trajectory Data



	Spatio-temporal Properties			Property Readings	
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) GPS Points Log



(b) An Example of Trajectory

Figure 2: An Example of Trajectory Data.

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) property readings, which may include the speed, direction, and any other information obtained at by the sensors. An entry in Figure 2a is an example of a GPS point, and the green dots on Figure 2b demonstrates the GPS point projection on the space.

Definition 2. *GPS Trajectory* A GPS trajectory Tr 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 $Tr = \{p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_5\}$ to form a trajectory.

Definition 3. *Map-Matched Trajectory* Trajectory map-matching projects the raw GPS points in a trajectory onto its corresponding spatial network. Figure 2b gives an map-matching example, where the green dots (GPS points) are projected onto the corresponding road segments (in red). Thus, the trajectory is converted as $Tr = \{s_1 \rightarrow s_2 \rightarrow s_3\}$.

2.2 Cloud Computing Components

In this subsection, we describe the main cloud computing components in Microsoft Azure: 1) cloud storage components and 2) cloud computing components.

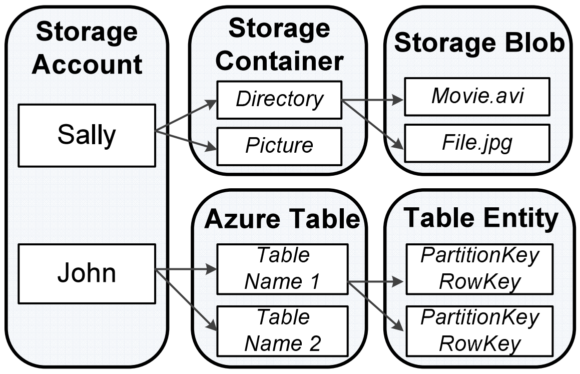


Figure 3: An Overview of Azure Storage.

Cloud Storage Components. The storage components in Azure [1] mainly contain Azure Blob, Azure Table, and Redis.

- **Azure Blob.** Blob (A Binary Large Object) is a collection of binary data stored as a single entity in Azure storage system.Blob can be compared to the file in the conventional file systems. Blob storage can store any type of text or binary data, such as a document, media file, or even application executable.
  Figure 3 gives an overview of the hierarchy structure of Blobs, where each storage account can create multiple storage containers, which have a grouping of a set of blobs (it is very similar to a directory in a conventional file system). A Blob file needs to be stored under a container. There are three types of blobs in Azure: 1) *block blob*, 2) *append blob* and 3) *page blob*. *Block blob* is ideal for storing text or binary files, such as documents and media files. *Append blobs* are optimized for append operations, which makes them useful for logging scenarios. *Page blobs* have a higher capacity, and are more efficient for frequent read/write operations. For example, Azure Virtual Machines use page blobs as OS and data disks.
  There is no limitation on how many files or containers can be created for a storage account. However, the total size of a storage account cannot exceed 500 TB [2]. Overall, Blobs are very useful in Azure to store unstructured data.
- **Azure Table** Azure Table stores structured NoSQL data in the cloud. Table storage is a key/attribute store with a schema-less design. Azure Table is essentially a cloud-based NoSQL database. Table storage is typically significantly lower in cost than traditional SQL for similar volumes of data, and Azure Table can be used in the cloud parallel computing frameworks, such as Hadoop, Spark and Storm. Figure 3 also gives the hierarchy structure of Azure Table, where each storage account can create unlimited number of tables (which are differentiated by table names). In each table, a table entities is identified by two keys, `partitionkey` and `rowkey`. Table entities with the same `partitionkey` are stored in the same partition.
  Azure Table is the best way to store the semi-structured datasets that don't require complex joins and foreign keys. The most efficient way to access Azure Table is through point queries (i.e., specify both the `partitionkey` and `rowkey`). Azure Table is also very efficient in answering the range queries of `rowkey` within the same `partitionkey`.
- **Azure Redis.** Azure Redis is based on the popular open-source Redis cache [3]. Azure Redis is an advanced in-memory key-value store, where keys can contain data structures such as strings, hashes, lists, sets, and sorted sets. Redis supports a set of atomic operations on these data types to ensure the data consistency.
  As Azure Redis stores the information in the memory of the Redis server, it is the best option to put the frequently accessed, and shared data in the system.

Cloud Computing Component. Besides the conventional virtual machines, Microsoft Azure also support the following distributed parallel computing platforms, in the Azure computing component, called HDinsight [4], to perform large scale data processing.

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

[2]https://azure.microsoft.com/en-us/documentation/articles/azure-subscription-service-limits/

[3]https://azure.microsoft.com/en-us/services/cache/

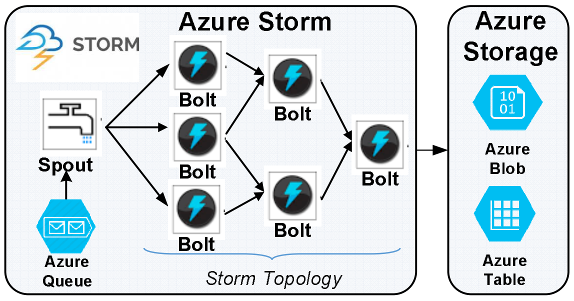


Figure 4: Storm Example in HDinsight.

- **Azure Hadoop** Hadoop is the most widely used Map-Reduce framework. Azure HDInsight deploys and provisions managed Apache Hadoop clusters in the cloud to process, analyze, and report on big data with high reliability and availability. HDInsight uses the Hortonworks Data Platform (HDP) Hadoop distribution. Hadoop framework is the best way to perform *offline batch-based* big data processing.
- **Azure Spark.** HDInsight includes Apache Spark, an open-source project in the Apache ecosystem that can run large-scale data analytics applications. Comparing to the conventional Hadoop framework, Spark avoids the disk I/Os and is able to deliver queries up to 100× faster. Spark is widely used in data analytics and machine learning areas, as it provides a common execution model for tasks like ETL, batch queries, interactive queries, real-time streaming, machine learning, and graph processing on data stored in Azure Storage.
- **Azure Storm.** Apache Storm is a distributed, real-time event processing solution for large, fast streams of data. It is the best option to process the real-time data and provide online services.
  Figure 4 gives an overview of the Storm platform. In a typical Storm platform, an Azure Queue is maintained to receive the real-time updates from different data sources. The *Spout* in Storm continuously reads the updates from the queue and distributes them to the *Bolts*. Each *Bolt* in the Storm may have different functionalities and be connected to each other based on the user's design, which is referred as a *Storm Topology*. Azure storage components, such as Blob and Table can be used in parallel throughout the *Storm topology*.

In our system, we use Storm to speed up the trajectory data process, because its ability to perform the real-time streaming and online services, which are very important in many urban computing applications

2.3 System Overview

Figure 5 gives a full picture of our trajectory data management system. As demonstrated at the top, the system provides an interface for more sophisticated urban data management/mining applications, by providing: 1) ID-temporal query, 2) Spatio-temporal query, and 3) map-matched results. To enable efficient and real-time service, the system employs three main modules, as follows:
Trajectory Storage. The trajectory storage module receives raw trajectory data from the user, which can be either historical files or real-time updates from the network. This module parses the raw

[4]https://azure.microsoft.com/en-us/services/hdinsight/

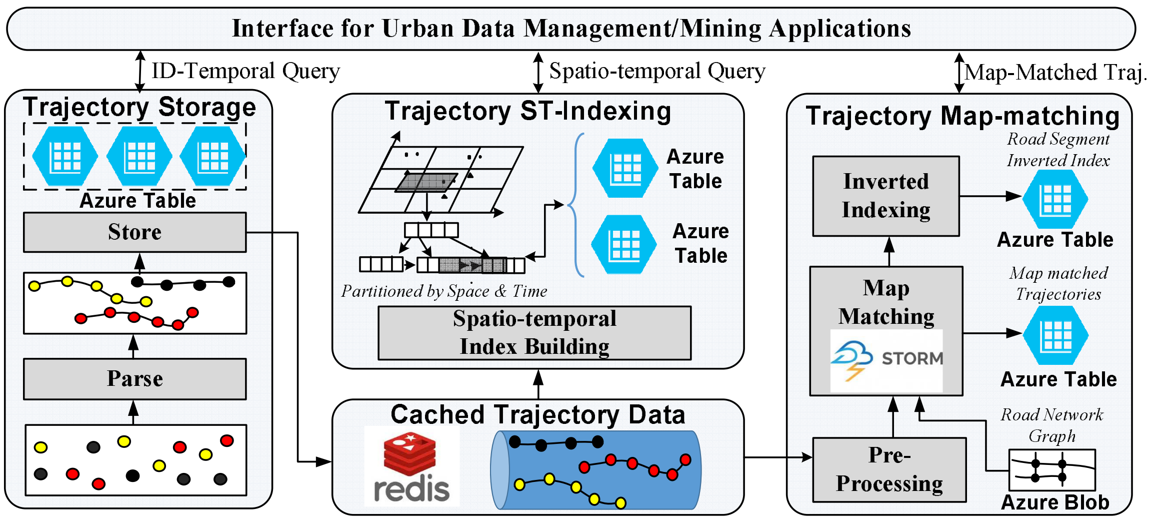


Figure 5: System Overview.

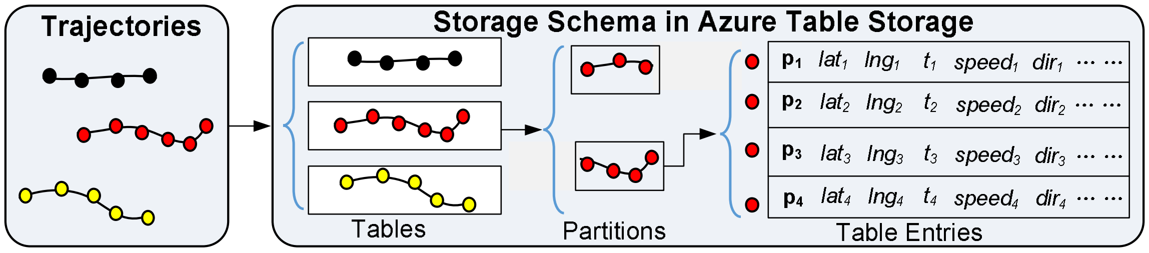

Trajectories
Storage Schema in Azure Table Storage
p1 lat1 lng1 t1 speed1 dir1 ··· ···
p2 lat2 lng2 t2 speed2 dir2 ··· ···
p3 lat3 lng3 t3 speed3 dir3 ··· ···
p4 lat4 lng4 t4 speed4 dir4 ··· ···
Tables
Partitions
Table Entries

Figure 6: Example Schema of Trajectory Storage in Azure Table.

GPS points, filters the error points, and organizes them into trajectories (i.e., grouped by plate numbers or trip IDs and ordered by time). Then, the processed trajectory data is stored in an Azure Table to support the ID-Temporal query (detailed in Section 3). Another important feature here is that we store an extra copy of organized trajectory data in a Redis server (i.e., the middle bottom component), because this data will be used by the other two modules and we can significantly reduce I/O cost by avoiding access to the disk.

Trajectory Spatio-temporal Indexing. The trajectory spatio-temporal indexing module takes the cached trajectory data from the Redis server, and organizes GPS points based on their spatio-temporal properties. A spatio-temporal index is built based on the Azure Table. In this way, the spatio-temporal range queries can be answered efficiently (detailed in Section 4).

Trajectory Map-matching. In this module, we also retrieve trajectory data from the Redis server, and map each GPS point in a trajectory onto the corresponding road segment. To enable efficiency and real-time service, we implement map-matching algorithms on the distributed streaming system, i.e., Storm. This module also maintains an inverted index for each road segment, to record the trajectory IDs passed to it (detailed in Section 5)

3. TRAJECTORY DATA STORE

In this section, we present details about the trajectory storage module. We first describe the overall process of storing trajectory data. After that, we demonstrate the process to support ID-temporal query efficiently.

Storage Process. There are two main processes in the storage module: 1) *trajectory data parsing* and 2) *trajectory data storing*.

• In the trajectory data parsing step, our system reads the raw trajectory data and organizes it. This step reads the raw data (i.e., essentially a set of unorganized GPS points) from the external data sources, e.g., from a file system in the offline scenario, or from a network stream in the real-time scenario. The main tasks in this step are: a) grouping the GPS points based on the trip IDs (or plate numbers), b) ordering points based on their timestamps, and c) filtering out error data. After these steps are completed, the organized trajectories are returned.

• In the trajectory storing step, the system gets the organized data and store it to the Azure storage component, including Azure Table (for historical access) and Redis server (for the most recent results).

Initially, we tried to store the trajectory data in the Azure blob and build an in-memory index, as it is a more straight-forward method. However, as we will latter demonstrate in the experiment section, the querying response time is much higher compare to Azure Table. Thus, in our system all of the trajectory data is stored in Azure table.

Figure 6 gives an overview of the storage schema we use in the system. To enable the computing parallelism on different trajectories, each trajectory is stored as a table in the system, where the table name is the plate number or trip ID. In this way, it not only gives semantic meanings to the storage schema, but also makes it possible to do parallel computing, if a query contains multiple trajectory IDs. In each table, the trajectory data is divided into different partitions, based on a system specified temporal range, e.g., one hour or one day. With different partition granularities, the system performance is different. For example, with a larger partition granularity, the insertion time may be reduced, as more entries can be inserted

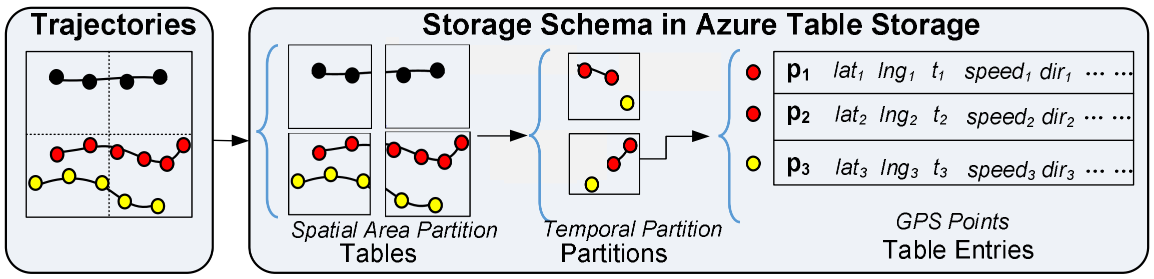

Trajectories
Storage Schema in Azure Table Storage
p1 lat1 lng1 t1 speed1 dir1 ··· ···
p2 lat2 lng2 t2 speed2 dir2 ··· ···
p3 lat3 lng3 t3 speed3 dir3 ··· ···
Spatial Area Partition
Tables
Temporal Partition
Partitions
GPS Points
Table Entries

Figure 7: Example Spatio-temporal Trajectory Schema in Azure Table.

together in a batch, while the query performance may suffer with more entries to scan in one partition. We will demonstrate more detailed performance tuning and provide design insights in Section 6. Finally, each GPS point in the partition is an entry in the table, where the `partitionkey` is the temporal partition, e.g., *"2016-06-25"* and the `rowkey` is the exact timestamp, like *"2016-06-25 11:24:43"*.

To minimize the I/O cost for the other modules, we also store a copy of recent processed trajectory data in a Redis server. The data stored in the server uses a dictionary data structure, where the `key` is the plate number or trip ID, while the `value` is the GPS points.

ID-Temporal Query Processing. In this step, the module answers ID-temporal query from the user with two scenarios:

• If the query asks for the most recent trajectory, our system answers it by retrieving content from the Redis server. In this case, the Redis server retrieves data based on the query ID (plate number or trip ID) and returns the contents to the user. In this way, we can avoid disk-related access and improve response time.

• If the query asks for the historical trajectory data. Our system first checks the parameters the of temporal range in the query. If the temporal range overlaps with multiple partitions, our system breaks the query into several small queries to each data partition and executes them in parallel.

4. TRAJECTORY ST-INDEXING

In this section, we present in details about the trajectory spatio-temporal indexing module. This module has two main components: 1) Spatio-temporal trajectory storage and 2) Spatio-temporal query processing.

Spatio-temporal Trajectory Storage. In this component, the system reads the cached trajectory data from the Redis server and store them into different spatio-temporal partitions. The main difference between the cloud-based data management system and the conventional spatio-temporal data management system, is that, instead of building an index to support the spatio-temporal query, our cloud-based system actually create an additional copy of the trajectory data and store them in the way that the spatio-temporal query can be executed efficiently. There are two main reasons to support our design: 1) the storage cost is much cheaper comparing to the computing in the cloud, e.g., it's less than 0.1 USD per 100TB/month [5]. And 2) the original trajectory data is stored in Azure Table, which is not efficient to access them in a random way, which may generated by a spatio-temporal query.

The main idea of the process is demonstrated in Figure 7. We first build a static spatial index over the trajectory data, e.g., equal-sized grids in our system. An Azure Table is created for each spatial partition in the index. Then, the partial trajectories that belong to the same spatial partition are grouped and inserted into the corresponding table. During that process, the partial trajectories are partitioned based on the timestamps of their GPS points, e.g., by one hour or one day. Finally, the GPS points are stored as table entries, where the `partitionkey` is the temporal range, e.g., *"2016-06-25"*, and the `rowkey` is the combination of timestamp and the trip ID, e.g., *"2016-06-25 10:22:32@Trip23112"*. The main reasons to design the `rowkey` in this way are: 1) simply using the timestamp as the `rowkey` may cause a conflict, where two GPS points in the same spatial region are generated at the same time; and 2) the `rowkey` still preserves the temporal information, which allows us to leverage the range query over `rowkey` to retrieve the GPS points in a temporal range efficiently.

Spatio-temporal Query Processing. To answer a spatio-temporal query using our system, we first examine the spatial range in the query to determine how many spatial partitions are covered. The system then retrieves data from different spatial partitions in parallel. For each retrieval process in a spatial table, the temporal range of the query is also broken into different temporal partitions and executed in parallel. Note here, although the `rowkey` in the spatio-temporal table contains both the timestamp and trip ID, we can still use the range `rowkey` search operation, as the Azure Table compares the `rowkey` strings from the left to right during the range query processing. Finally, results are returned by aggregating all data obtained from different spatial table and temporal partitions.

As a result, it is also very clear that the granularities of spatial partitions and temporal partitions significantly impact the efficiency of the data storage and query process, which we will demonstrate in detail in the experiment section.

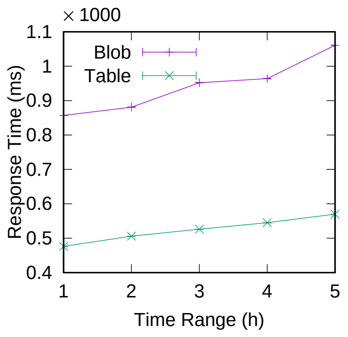
5. TRAJECTORY MAP-MATCHING

Trajectory map-matching projects the raw GPS points in a trajectory on a spatial network, converting a sequence of latitude/longitude coordinates to a sequence of road segment IDs. Map-matching is a vital and fundamental process for many urban applications, e.g., traffic speed/volume inference, vehicle navigation, and destination inference.

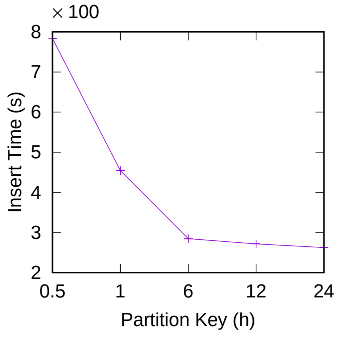
In supporting these urban applications online, it is very challenging to provide a map-matching service in real-time. This is for two primary reasons: 1) the volume of traffic is huge, often resulting in up to tens of thousands of real-time trajectory updates in the system with each trajectory update containing around ten GPS points; and 2) the task of map-matching itself is very time-consuming, e.g., it took us two weeks to perform the map-matching task on two months' trajectory data containing 3,500 taxis from the city of Tianjin over a server computer with Intel Xeon E5 CPU @ 2.4GHz processor, and 128 GB RAM [11].

To support the map-matching service efficiently and online, we leverage the distributed streaming computing platform in Azure, i.e., Storm, to speed up the map-matching process. Figure 8 gives an overview of the Storm topology in our system, which includes

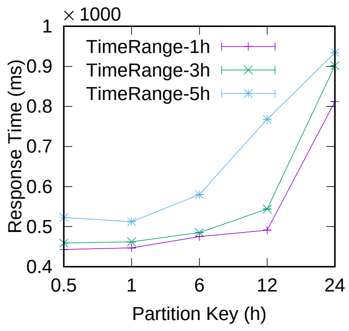
[5]https://azure.microsoft.com/en-us/pricing/details/storage/



(a) Blob VS. Table.



(b) Insertion wrt. Partition Sizes.



(c) Response wrt. Partition Sizes.

Figure 9: Trajectory Store & ID-Temporal Query Experiments.

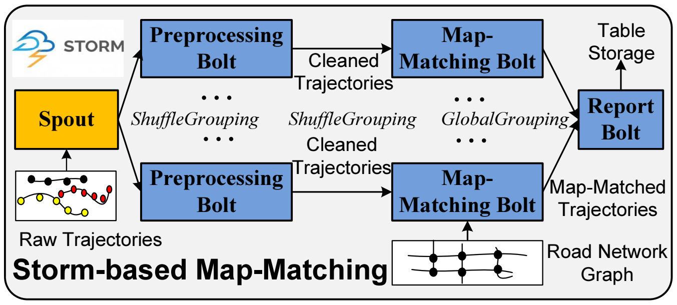


Figure 8: Storm Topology of Map-matching Pipeline.

four components:

- *Spout.* The spout reads raw trajectories from the Redis server (in the real-time scenario) or Azure Table (in the offline scenario), and then emits the raw trajectories to the preprocessing bolts using the shuffle grouping mechanism.
- *Preprocessing Bolt.* The preprocessing bolt filters the noisy GPS points from the raw trajectory to remove the outlier points. In this bolt, we use a heuristics-based outlier detection method [1]. After that, each cleaned trajectory is emitted to map-matching bolt using shuffle grouping mechanism.
- *Map-Matching Bolt.* The map-matching bolt takes the cleaned trajectory and the road network and maps each GPS point onto the corresponding road segment. In this bolt, we use an interactive-voting based map matching algorithm [12], as it does not only consider the spatial and temporal information of a GPS trajectory but also devises a voting-based strategy to model the weighted mutual influences between GPS points. After the map-matching task is complete, the result is emitted to the report bolt using global grouping mechanism.
- *Report Bolt.* The report bolt caches the map-matched trajectories and writes the results in batch to a Redis server or Azure Table to reduce the high I/O overhead of frequent small writings.

The advantages to parallel map-matching computing at the trajectory level in our framework, are: 1) it makes the system more flexible, as it is much easier to transplant any other map-matching algorithms in our system. We only need to import the algorithm into the map-matching bolt, and all the inputs and outputs are the same. 2) It is much easier for the spout to distribute the workload to different bolts, as the trajectories received in the spout are processed by the storage module and are always separated by vehicle IDs or trip IDs.

After the map-matching process is done for the trajectories, the trajectory data is converted into a sequence of road segment IDs. Our system also builds an inverted index for each road segment to store the IDs and corresponding timestamps of the trajectories that have passed it. In this way, we are able to answer the temporal query like, "what trajectories have passed road segment r_i yesterday afternoon?".

6. EXPERIMENT EVALUATION

In this section, we provide a set of experiments to 1) demonstrate the efficiency of our trajectory data management system, and 2) share some lessons we learn during the parameter tuning.

6.1 Trajectory Storage

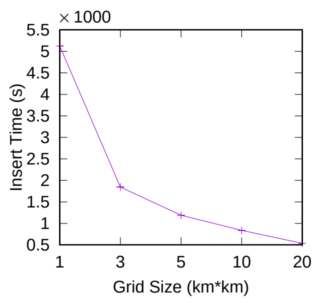
The first set of experiments demonstrates the efficiency of the trajectory storage module. We first show the performance differences between Azure Blob and Azure Table. We then show the efficiency differences on data insertion and querying with different `partitionkey` settings in Azure Table.

Efficiency of Blob & Table. Figure 9a gives the performance of the ID-temporal query on Blob storage and Table storage, while increasing the query time range. Each Blob file and Azure partition contains one-day trajectory information. There are two insights we can get in this figure: 1) query performance is better when the query range is smaller, as more data is accessed and transferred with a larger temporal range; and 2) Azure Table performs much faster(over 2 times faster) and more consistently than the Azure Blob. As a result, we use Azure Table to store the trajectories.

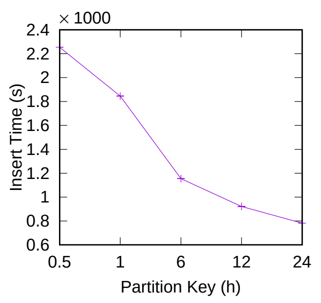
Data Storage Performance. In this experiment, we test the insertion performance with different granularities of `partitionkey`, from half hour to one day. In this experiment, 100 one-day trajectories are inserted. We can see from the figure that with a larger `partitionkey` the insertion performance increases significantly first, and then becomes more consistent. This is because with a large partition key, more data can be inserted into one batch. On the other hand, when the `partitionkey` gets larger the performance bottleneck changes per network communication.

Different Temporal Ranges. Figure 9c gives the performance ID-temporal query evaluation with different granularities of the `partitionkey`. We can see that in general, the data schema with smaller partition performs better, as in the large partition the cloud storage the system needs to scan more table entities to retrieve query results.

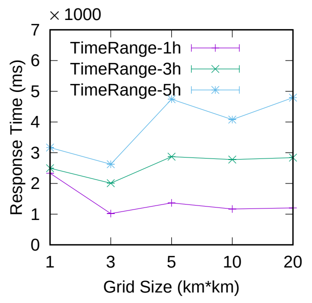
As a result, there is a trade-off in choosing the size of the `partitionkey`, where a large partition performs better in data storage and the loading phase. However, the query performance



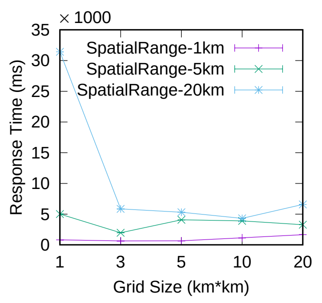
(a) Insertion wrt. Grid Sizes.



(b) Insertion wrt. Temporal Sizes.

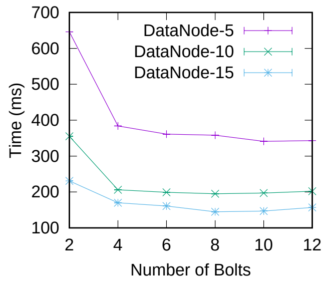


(c) Query wrt. Temporal Ranges.

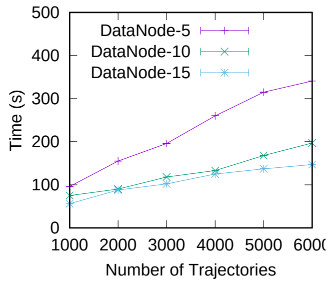


(d) Query wrt. Spatial Ranges.

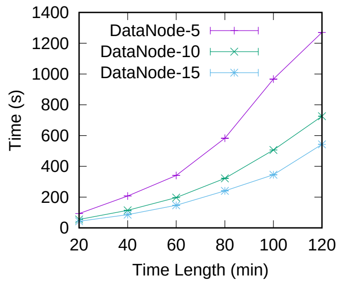
Figure 10: Spatio-temporal Indexing & Query Processing.



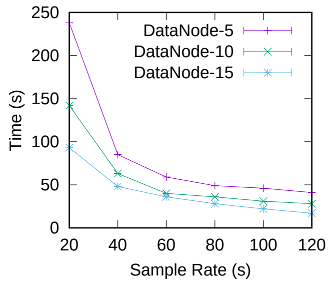
(a) Bolts Per Worker.



(b) Trajectory Size.



(c) Trajectory Length.



(d) Trajectory Sample Rate.

Figure 11: Storm-based Map-Matching Experiments.

decreases significantly with large partitions.

6.2 Trajectory ST-Indexing

This set of experiments illustrates the efficiency of the trajectory spatio-temporal indexing module. We first demonstrate the system efficiency on indexing the trajectory data with different spatial and temporal granularities. Then, we evaluate the spatio-temporal query performances with different query parameters.

Trajectory Data Indexing. Figure 10a gives the data insertion performance when we divide the spatial area into different sized grid cells, and the `partitionkey` is set as one hour. We can see that insertion performance is better when the grid size is bigger, as more trajectories can be inserted into one batch. Figure 10b demonstrates the efficiency on a different `partitionkey`, where the grid size is fixed as 3 km^2. Similarly, the insertion time decreases, when the partition is bigger.

Spatio-temporal Query Processing. Figure 10c g gives the efficiency results of spatio-temporal queries with different temporal ranges and the same spatial range (i.e., 3 km^2). The `partitionkey` is set as one hour. In this experiment, the response time first decreases, as in 1 km^2 grid setting, more tables need to be accessed to answer the query. On the other hand, the performance generally increases with larger grid sizes, as there are more candidate trajectories to be filtered in each grid cell. Also, the query with a larger temporal range has a higher response time, as more partitions need to be accessed and more data is transferred via the network.

Figure 10d gives the efficiency evaluation of spatio-temporal queries with different spatial areas and the same temporal range (i.e., one hour). It is clear that the query with a smaller spatial range performs better, as less data is accessed and transferred. It is also interesting to see that the queries usually perform the best when the grid size is similar to the spatial range size in the query. This is because if the grid size is smaller, more tables need to be accessed. On the other hand, if the grid size is bigger, more data needs to be filtered during the process.

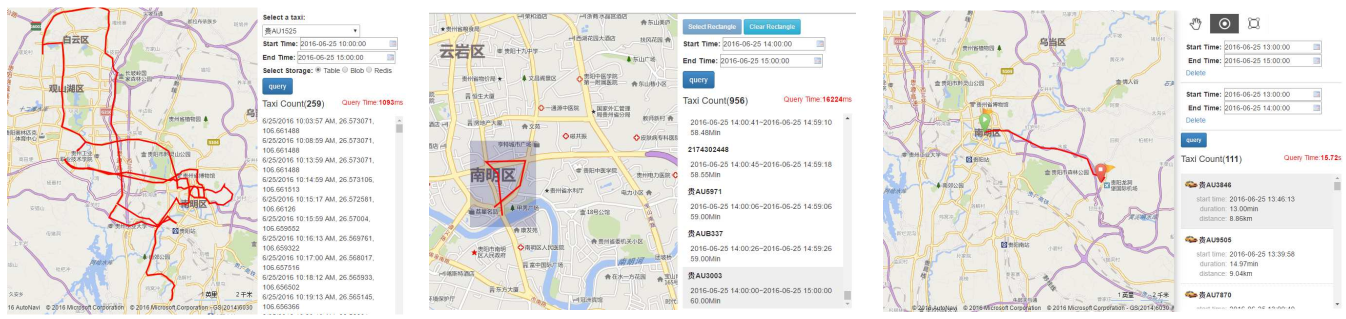
As a result, in order to serve the spatio-temporal queries and insertions in a more efficient way, there are many factors that need to be considered, such as the most frequent spatial and temporal range of the query and the granularity of the batch insertion sizes.

6.3 Storm-based Map-Matching

In this set of experiments, we provide a set of experiments on the efficiency of the map-matching process over the Storm platform. We first test the settings in Storm to find the optimal parallelism settings. After that, we provide a set of the map-matching efficiency experiments on different sizes of cloud clusters, with respect to different trajectory sizes, average lengths, and sample rates.

Parallelism Degree. When designing the parallelism of Storm topology, there are three important parallelism factors, a user needs to specify: 1) the number of data nodes, which is essentially the size of computing clusters in the cloud, with each node containing a 4-core CPU. 2) the number of workers, which can be analogized as a computing process. And 3) the number of bolts, which is the number of threads running in one process (i.e., worker in this context). It is clear that to fully utilize the computing power of a cluster, the number of workers should be greater than the number of data nodes in Storm. However, the optimal parameter of bolts per worker is highly dependent on the application scenario.

Figure 11a presents the results on our map-matching module using Storm. In each experiment, we perform the map-matching task on 6000 trajectories with an average length of one hour. The experiments are done with a different number of bolts per worker with three different cluster sizes (i.e., 5, 10 and 15). It is clear in the figure that the cluster with a higher number of data nodes always performs better. Moreover, the efficiency of the system improves significantly when we change two bolts per worker to four bolts per worker. This is because each CPU in our cluster has four cores. In


Select a taxi:
贵AU1525
Start Time: 2016-06-25 10:00:00
End Time: 2016-06-25 15:00:00
Select Storage: [✓ Table] Blob Redis
query
Taxi Count(**259**) Query Time:**1093**ms
6/25/2016 10:03:57 AM, 26.573071, 106.661488
6/25/2016 10:08:59 AM, 26.573071, 106.661488
6/25/2016 10:13:59 AM, 26.573071, 106.661488
6/25/2016 10:14:59 AM, 26.573106, 106.661513
6/25/2016 10:15:17 AM, 26.572581, 106.66126
6/25/2016 10:15:59 AM, 26.57004, 106.659552
6/25/2016 10:16:13 AM, 26.569761, 106.659322
6/25/2016 10:17:00 AM, 26.568017, 106.657516
6/25/2016 10:18:12 AM, 26.565933, 106.656502
6/25/2016 10:19:13 AM, 26.565145, 106.656366

(a) Example of Plate Temporal Query

Select Rectangle Clear Rectangle
Start Time: 2016-06-25 14:00:00
End Time: 2016-06-25 15:00:00
query
Taxi Count(**956**) Query Time:**16224**ms
2016-06-25 14:00:41~2016-06-25 14:59:10
58.48Min
2174302448
2016-06-25 14:00:45~2016-06-25 14:59:18
58.55Min
贵AU5971
2016-06-25 14:00:06~2016-06-25 14:59:06
59.00Min
贵AUB337
2016-06-25 14:00:26~2016-06-25 14:59:26
59.00Min
贵AU3003
2016-06-25 14:00:00~2016-06-25 15:00:00
60.00Min

(b) Example of Spatio-temporal Query

Start Time: 2016-06-25 13:00:00
End Time: 2016-06-25 15:00:00
Delete
Start Time: 2016-06-25 13:00:00
End Time: 2016-06-25 14:00:00
Delete
query
Taxi Count(**111**) Query Time:**15.72**s
贵AU3846
start time: 2016-06-25 13:46:13
duration: 13.00min
distance: 8.86km
贵AU9505
start time: 2016-06-25 13:39:58
duration: 14.97min
distance: 9.04km
贵AU7870

(c) Example of OD-based Query

Figure 12: Real-time Taxi Data Management System.

the two bolts' setting, the parallel capability is not fully exploited. On the other hand, the map-matching efficiency decreases when we put more than eight bolts per worker in the system. This is because more bolts introduces overhead during context switching. As a result, to achieve a reasonable parallelism in Storm, the user should have a number of bolts per worker that is at least the same as the number of cores in one data node in the cluster.

In the remaining experiments, we set the number of bolts per worker as eight.

Different Number of Trajectories. In this experiment, we test the efficiency of the map-matching module with different numbers of trajectories. Figure 11b 11b illustrates the performance, where with more trajectories the processing time increases. One interesting insight here is that with a total number of trajectories that is less than 4,000, the performance improvement of the 15-data node cluster is very limited. However, an extra five data nodes would cost more than 2,500 USD per month. Hence, in that scenario, a 10-node cluster is a more economical solution.

Different Length of Trajectories. In this experiment, we present map-matching efficiency with different trajectory lengths, where each experiment performs map-matching for 6,000 trajectories. It is clear from Figure 11c that with longer trajectories, map-matching takes more time and the bigger cluster has a lower processing time.

Different Trajectory Sample Rates. In Figure 11d, we present the map-matching efficiency with different trajectory sample rates (the average temporal intervals between consecutive GPS points), where each experiment performs map-matching for 1,000 one-hour trajectories. We can see that with trajectories with less interval time take more time, as there are more points to perform map-matching.

7. CASE STUDIES

With the availability of the trajectory data management system in Azure, many urban applications can be built with much less effort. In this section, we present three case studies we build using our system: 1) a real-time taxi data management system, 2) a real-time traffic data modeling system, and 3) a resource allocation system based on trajectories.

7.1 Real-time Taxi Data Management System

The taxi data management system we build based on real-time taxi trajectory updates from 7,500 taxi cabs in the city of Guiyang, China. In this system, we create an Azure Cloud Service to query real-time taxi trajectories and return the result in JSON format at the back-end. A web interface is built as an Azure Web App as the front-end, which supports three functions:

Plate Temporal Query. It is used to find out what did a taxi travel in a given time range. As demonstrated in Figure 12a,the user can specify a taxi plate number and a temporal range, the system will retrieve the trajectory data of the taxi within the given temporal period. To improve the response time for real-time usage, the system returns the most recent one hour result from the Redis server directly, while the historical results are read from the Azure table.

Spatio-temporal Query. This is used to get the taxi cabs and their trips that fall in a given spatio-temporal range, which is very useful for users booking taxi services. As shown in Figure 12b, the user can draw a rectangle on the map as the spatial range, and input a temporal range from the box. Then, the system returns all taxi cabs and their trajectories within the specified spatio-temporal range.

OD-based Query. This query the number of trips the taxis make, given an OD (i.e., origin and destination) pair. This is very useful for finding a taxi if a passenger lost something in a cab, and can only pinpoint his/her origin and destination. As demonstrated in figure 12c, the user can select two locations as the origin and destination and two time windows corresponding to each location. The system returns all qualified trips, where essentially the back-end system executes two spatio-temporal queries to find the plate numbers and the times-tamps, and then executes another plate-temporal query to retrieve detailed trajectories.

7.2 Real-Time Traffic Modeling System

The real-time traffic modeling system aims to provide three important pieces of information on each road segment in a city: 1) travel speed, 2) traffic volume, and 3) emission levels. To be able to infer any of these three bits of information, it is vital to be able to perform map-matching in real-time. Our system constantly gets real-time trajectory updates from over 7,500 taxi and performs all tasks in less than one minute, using a 30-node Storm system in Microsoft Azure.

Of the three aforementioned three tasks, travel speed calculation is the most fundamental one. After we get the map-matched trajectories, we can calculate the average travel speed for the road segments covered by the trajectory data received. As shown in Fig-

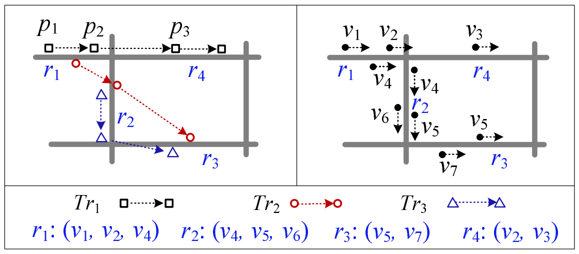
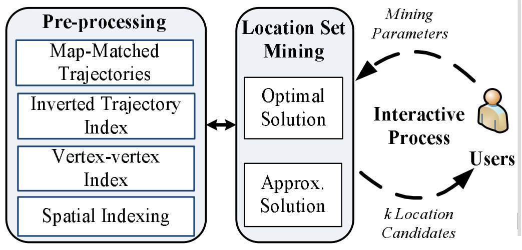
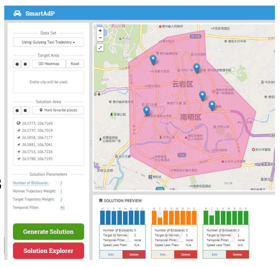


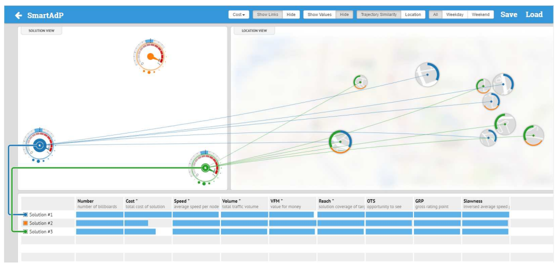
Figure 14: Travel Speed Inference.



(a) System Overview



(b) Solution Generation



(c) Solution Explorer

Figure 13: Trajectory-based Resource Allocation System.

ure 14, three vehicles traveled four road segments r_1, r_2, r_3 and r_4, generating three trajectories Tr_1, Tr_2, and Tr_3.map-matching, each point from a trajectory is mapped onto a road segment. We can then calculate the travel speed for each point based on Equation 1.

$$v_1 = \frac{Dist(p_1.l, p_2.l)}{|p_2.t - p_1.t|} \tag{1}$$

where Dist is a function calculating the road network distance between two points. Thus, we can compute the average travel speed of a road segment as Equation 2.

$$\overline{v} = \Sigma_i^m \frac{v_i}{n} \tag{2}$$

After that, we can infer the travel speed of the road segment without any trajectories using a matrix decomposition model [5]. We can then infer the traffic volume and emission levels of each road segment using a graphic model [13]. As illustrated in Figure 15, we build a real-time traffic modeling system using the calculated travel speed information. The system models real-time traffic conditions on each road segment and infers city-wild gas consumption and pollution emissions of vehicles.

7.3 Trajectory-based Resource Allocation

The trajectory-based resource allocation system finding a set of k locations on a road network, which, collectively, traversed by the maximum number of unique trajectories in a given region. This application is vital to many resource allocation applications, like advertisement placement, charging/gas station placement, and chain business location selection.

As shown in Figure 13a, the system requires a large set of map-matched trajectories, as well as the inverted trajectory index, which are generated by our trajectory data management system. As a result, we are able to build an interactive visual analytic system [14, 15] that supports solution generation and solution explorer.

Solution generation. As shown in Figure 13b,a user can select

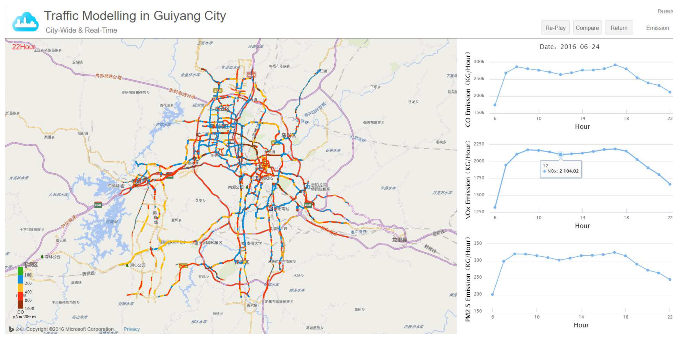


Figure 15: Demonstration of Traffic Modeling System.

an arbitrary spatial range to place the stations/advertisements and specify the number of stations/advertisement needed. The system will automatically mine the underlying trajectory dataset and provide a solution, e.g., with 5 location suggestions. The user can interact with the solution to add or remove some unqualified locations to generate new solutions, like the ones on the bottom.

Solution explorer. Figure 13c demonstrates the solution explorer, where the system provide more different views to help the user to select the most suitable location combinations.

8. RELATED WORKS

In this section, we present the related research in the following three aspects: 1) trajectory querying & mining, 2) trajectory map-matching, and 3) parallel spatial computing platforms.

Trajectory Query & Mining. The availability of location-acquisition techniques has enabled the collection of massive trajectory data. With the massive trajectory data, many different spatio-temporal indexes have been proposed to support spatio-temporal queries, detailed in [16, 17]. With these efficient indexes and accessing methods, different trajectory-based mining applications emerge [1][18][19][20]. [18] studies the problem of discovering the gathering patterns from trajectories. [19] proposes to estimate the travel-time of a path in real time in a city based on the GPS trajectories of vehicles. [20] studies a query which finds the most frequent path of user-specified source and destination from historical trajectories. However, all of the existing accessing methods and mining applications are implemented in a centralized way. In our system, we leverage the existing indexes in Azure storage and make them support the spatio-temporal query efficiently. The most similar industrial work are [21, 22], which build spatial index over the NoSQL database. However, these work only support spatial objects, rather than trajectories.

Trajectory Map-Matching. Map-matching is a very important function in our system. The existing map-matching algorithms can be categorized based on their models in to four groups [1]: 1) geometric-based [23], 2) topological-based [24], 3) probabilistic-based approaches [25] and 4) other advanced techniques [12]. However, the goal in our system is not proposing a new map-matching algorithm, but to speed the service up using parallel streaming system to support the real-time urban applications. The closest work with us is [26], which uses Hadoop framework to parallel the map-matching algorithms. However, in our system Storm is a more suitable platform, as it is able to handle the streaming GPS data and provide the real-time service.

Parallel Spatial Computing Platforms. The first attempt to involve Hadoop in spatial computing is done by Parallel SECONDO [27], which combines Hadoop with SECONDO. Hadoop-GIS [7] utilizes global partition indexing and customized on-demand local spatial indexing to efficient supports multiple types of spatial queries. SpatialHadoop [8] is comprehensive extension to Hadoop, which has native support for spatial data by modifying the underlying code of Hadoop. Most recently, due to the high IO cost in Hadoop, there are some systems, e.g., SpatialSpark [9] and GeoSpark [10], try to support large scale spatial queries and joins in Spark framework. However, all of above systems only support spatial query and mining, which cannot directly support spatio-temporal data, such as trajectories. The closest work to us is [28], which can answer trajectory spatio-temporal range queries using Hadoop framework. Compare to [28], our system supports more query types and has the ability to handle the real-time trajectory updates.

9. CONCLUSION

This work presents a holistic cloud-based trajectory data management system, which serves as the foundation of many urban applications. The system is able to solve ID-Temporal queries, spatio-temporal queries, and trajectory map-matching in an efficient and real-time way. Our system leverages the storage components in Azure, including Azure Blob, Azure Table, and Azure Redis. We design and implement the storage component in a way that supports queries more efficiently. Also, we utilize the Apache Storm, the distributed streaming system in Azure to perform efficient and real-time map-matching service.

We perform extensive experiments on real trajectory data. We share details of the design and implementation of the system, which we believe are very useful for shifting the spatio-temporal computing and mining to the cloud. Our system has been implemented in in many real urban computing applications internally. Our system significantly improves efficiency and usability. For future work, we will test more data-driven indexes such as R+ trees and Quad-tree, as well as support more types of trajectory queries and operations. Ultimately, we would like to incorporate the trajectory computing as a standard service in Microsoft Azure Cloud.

10. REFERENCES

[1] Y. Zheng, "Trajectory data mining: an overview," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 3, p. 29, 2015.

[2] Y. Zheng, L. Capra, O. Wolfson, and H. Yang, "Urban computing: concepts, methodologies, and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 5, no. 3, p. 38, 2014.

[3] Y. Zheng, F. Liu, and H.-P. Hsieh, "U-air: when urban air quality inference meets big data," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 1436–1444.

[4] 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 *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 2267–2276.

[5] Y. Wang, Y. Zheng, and Y. Xue, "Travel time estimation of a path using sparse trajectories," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 25–34.

[6] A. M. Hendawi, J. Bao, M. F. Mokbel, and M. Ali, "Predictive tree: An efficient index for predictive queries on road networks," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 1215–1226.

[7] 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," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.

[8] A. Eldawy and M. F. Mokbel, "A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1230–1233, 2013.

[9] S. You, J. Zhang, and L. Gruenwald, "Spatial join query processing in cloud: Analyzing design choices and performance comparisons," in *Parallel Processing Workshops (ICPPW), 2015 44th International Conference on*. IEEE, 2015, pp. 90–97.

[10] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2015, p. 70.

[11] Y. Li, Y. Zheng, S. Ji, W. Wang, and Z. Gong, "Location selection for ambulance stations: a data-driven approach," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2015, p. 85.

[12] J. Yuan, Y. Zheng, C. Zhang, X. Xie, and G.-Z. Sun, "An interactive-voting based map matching algorithm," in *Proceedings of the 2010 Eleventh International Conference on Mobile Data Management*. IEEE Computer Society, 2010, pp. 43–52.

[13] J. Shang, Y. Zheng, W. Tong, E. Chang, and Y. Yu, "Inferring gas consumption and pollution emission of vehicles throughout a city," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 1027–1036.

[14] D. Liu, D. Weng, Y. Li, Y. Wu, J. Bao, Y. Zheng, and H. Qu, "SmartAdP: Visual Analytics of Large-scale Taxi Trajectories for Selecting Billboard Locations," in *The IEEE Conference on Visual Analytics Science and Technology (IEEE VAST 2016)*. IEEE Computer Society, 2016.

[15] Y. Li, J. Bao, Y. Li, Y. Wu, Z. Gong, and Y. Zheng, "Mining the Most Influential k-Location Set from Massive Trajectories," in *SIGSPATIAL*. ACM, 2016.

[16] 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.

[17] L.-V. Nguyen-Dinh, W. G. Aref, and M. Mokbel, "Spatio-temporal access methods: Part 2 (2003-2010)," 2010.

[18] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang, "On discovery of gathering patterns from trajectories," in *ICDE*, 2013, pp. 242–253.

[19] Y. Wang, Y. Zheng, and Y. Xue, "Travel time estimation of a path using sparse trajectories," in *SIGKDD*, 2014, pp. 25–34.

[20] W. Luo, H. Tan, L. Chen, and L. M. Ni, "Finding time period-based most frequent path in big trajectory data," in *SIGMOD*, 2013, pp. 713–724.

[21] "Geocouch," https://github.com/couchbase/geocouch/.

[22] "neo4j/spatial," https://github.com/neo4j/spatial/.

[23] J. S. Greenfeld, "Matching gps observations to locations on a digital map," in *Transportation Research Board 81st Annual Meeting*, 2002.

[24] H. Yin and O. Wolfson, "A weight-based map matching method in moving objects databases," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, 2004, pp. 437–438.

[25] O. Pink and B. Hummel, "A statistical approach to map matching using road network geometry, topology and vehicular motion constraints," in *2008 11th International IEEE Conference on Intelligent Transportation Systems*. IEEE, 2008, pp. 862–867.

[26] J. Huang, S. Qiao, H. Yu, J. Qie, and C. Liu, "Parallel map matching on massive vehicle gps data using mapreduce," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC)*. IEEE, 2013, pp. 1498–1503.

[27] J. Lu and R. H. Güting, "Parallel secondo: boosting database engines with hadoop," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. IEEE, 2012, pp. 738–743.

[28] 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.