TrajMesa: A Distributed NoSQL Storage Engine for Big Trajectory Data

Ruiyuan Li[1,2], Huajun He[3,2], Rubin Wang[3,2], Sijie Ruan[1,2], Yuan Sui[2], Jie Bao[2], Yu Zheng[1,2]

[1] *Xidian University, China* [2] *JD Intelligent Cities Research, China* [3] *Southwest Jiaotong University, China*

{ruiyuan.li, hehuajun3, wangrubin3, ruansijie, suiyuan, baojie3, zhengyu}@jd.com

***Abstract*—Trajectory data is very useful for many urban applications. However, due to its spatio-temporal and high-volume properties, it is challenging to manage trajectory data. Existing trajectory data management frameworks suffer from scalability problem, and only support limited trajectory queries. This paper proposes a holistic distributed NoSQL trajectory storage engine, TrajMesa, based on GeoMesa, an open-source indexing toolkit for spatio-temporal data. TrajMesa adopts a novel storage schema, which reduces the storage size tremendously. We also devise novel indexing key designs, and propose a bunch of pruning strategies. TrajMesa can support plentiful queries efficiently, including ID-Temporal query, spatial range query, similarity query, and k-NN query. Experimental results show the powerful query efficiency and scalability of TrajMesa.**

I. Introduction

With the proliferation of positioning technology, a large number of trajectories have been generated. To utilize such huge trajectories, various trajectory queries have been proposed: 1) **ID temporal query**, which retrieves the trajectories of a given moving object within a specified time range, is used frequently in package tracking services; 2) **spatial range query**, which finds the trajectories travelling through a given spatial range, can be used to discover reachable areas [1]; 3) **similarity query**, which returns the trajectories similar to a given trajectory, would help police to detect illegal parking in streets [2]; and 4) **k-NN (Nearest Neighbour) query**, which finds k trajectories that are most similar to a given trajectory, can investigate driving habits [3].

It is desirable for a scaleable unified trajectory storage engine to support all of these queries efficiently. Centralized solutions [4] are based on a single machine, thus could not cope with such huge trajectories obviously. Most recently, there have emerged many distributed in-memory trajectory data management frameworks, e.g, [5, 6]. However, they still suffer from several limitations. First, these frameworks load all trajectories into memory. They require high-performance clusters with much memory, hence their extensibility is limited. Second, for each request, they need to scan big indexes in memory, which is costly. Third, all of these frameworks only support very limited trajectory queries, therefore cannot support plentiful urban applications. Distributed NoSQL (Not Only SQL) data stores, e.g., HBase [7], are suitable for read/write random access to big data. However, due to lack of secondary indexes, NoSQL data stores do not natively support spatio-temporal data management. GeoMesa [8] is an open-source tool that manages large-scale spatio-temporal data on the top of distributed NoSQL data stores. It transforms multi-dimensional information into one-dimensional key. However, GeoMesa cannot be applied to manage trajectories directly.

This work is supported by NSFC (Grant No. 61672399, No. U1609217).

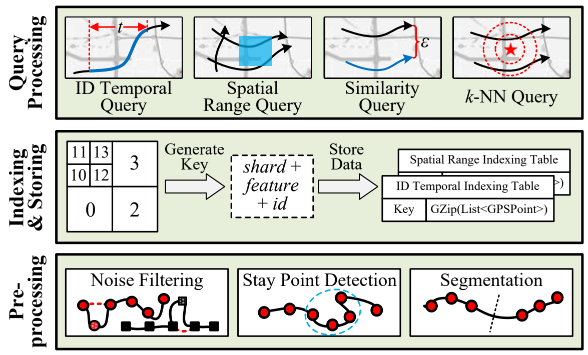


Fig. 1. Framework of TrajMesa.

This paper proposes a holistic distributed NoSQL trajectory storage engine, **TrajMesa**, based on GeoMesa. As shown in Figure 1, TrajMesa incorporates three standard processes:

(1) **Preprocessing**, which performs three main tasks: noise filtering, stay point detection, and segmentation. Trajectory preprocessing is not only necessary for many urban applications, but also very important for our selection of the underlying storage schema and index building. More details can be found in our previous work [9].

(2) **Indexing & Storing**, which builds indexes for the preprocessed trajectories, and stores the trajectory data through GeoMesa. Specifically, we generate two different keys that combine the spatio-temporal and other necessary information of a trajectory. Each key and the trajectory data forms a key-value pair, which is then stored into the key-value data store of GeoMesa. In other words, we store two copies of a trajectory into two tables with different keys (detailed in Section III).

(3) **Query Processing**. With the help of built indexes, TrajMesa efficiently supports most useful trajectory queries, including: ID temporal query, spatial range query, similarity query, and k-NN query (detailed in Section IV).

The contributions of this paper are summarized as follows:

(1) We take the first attempt to build a holistic distributed NoSQL trajectory storage engine based on GeoMesa, in which a novel trajectory storage schema is designed.

(2) We devise novel indexing key schemas and multiple pruning strategies for various trajectory queries.

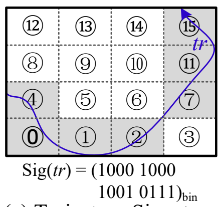
(3) Experiments show the powerful query efficiency and scalability of TrajMesa.

Key	Value	
Point Key	Spatio-temporal Properties	Other Properties
p_1	lat_1 lng_1 t_1	oid_1 tid_1 ⋯
p_2	lat_2 lng_2 t_2	oid_2 tid_2 ⋯
⋯	⋯	⋯
p_n	lat_n lng_n t_n	oid_n tid_n ⋯

(a) Vertical Storage Schema

Key	Value			
Traj. Key	Spatio-temporal Properties	GPS Point List	Signature	Other Properties
$traj_1$	mbr_1 t_{s1} t_{e1} p_{s1} p_{e1}	gzip(kryo(List<GPSPoint>))	00010101	oid_1 tid_1 ⋯
$traj_2$	mbr_2 t_{s2} t_{e2} p_{s2} p_{e2}	gzip(kryo(List<GPSPoint>))	10011001	oid_2 tid_2 ⋯
⋯	⋯	⋯	⋯	⋯
$traj_m$	mbr_m t_{sm} t_{em} p_{sm} p_{em}	gzip(kryo(List<GPSPoint>))	10001001	oid_m tid_m ⋯

(b) Horizontal Storage Schema



(c) Trajectory Signature

Fig. 2. Storage Schema Selection.

II. Preliminary

Definition 1. (Trajectory) A trajectory $tr = \{p_1 \to p_2 \to ... \to p_n\}$ is a list of GPS points ordered by their timestamps, where $p_i = \{lat_i, lng_i, t_i\}$ is a GPS point.

Definition 2. (ID Temporal Query) Given a dataset $\mathcal{T}$, a moving object oid, a time range $R = [t_s, t_e]$, ID temporal query returns $tr_i \in \mathcal{T}$, where $tr_i.oid = oid$, and there exists at least one GPS point p_j in tr_i that is generated during R.

Definition 3. (Spatial Range Query) Given a dataset $\mathcal{T}$, a spatial range $S = \{lat_{min}, lng_{min}, lat_{max}, lng_{max}\}$, spatial range query returns $tr_i \in \mathcal{T}$, where there exists at least one GPS point p_j in tr_i that is located in S.

Definition 4. (Similarity Query) Given a dataset $\mathcal{T}$, a query trajectory q, a distance function f, a distance threshold ε, similarity query finds $tr_i \in \mathcal{T}$, where the distance between q and tr_i is not greater than ε.

We now support two common trajectory distance functions, Fréchet distance [10] f_F and Hausdorff distance [11] f_H.

Definition 5. (k-NN Query) Given a dataset $\mathcal{T}$, a query q, a positive integer k, a distance function f, k-NN query returns a set of trajectories $\mathcal{T}' \subseteq \mathcal{T}$, where $|\mathcal{T}'| = k$, and for each $tr_i \in \mathcal{T}'$, $tr_j \in \mathcal{T} \setminus \mathcal{T}'$, $f(q, tr_i) < f(q, tr_j)$.

If q is a trajectory, f can be f_F or f_H, and it is called k-NN trajectory query [5, 12]. If q is a point, f can be defined as Equ (1), and it is entitled k-NN point query [12].

$$f_P(q, tr) = \min_{p_j \in tr} d(q, p_j) \tag{1}$$

where $d(q, p_j)$ is the Euclidean distance between q and p_j.

III. Indexing And Storing

A. Storage Schema Selection

Vertical Storage Schema. One basic idea to store trajectories in a key-value store is that, the trajectory data is stored with each GPS point as one row, just as most cloud-based trajectory management systems did [9, 13]. We call this schema as vertical storage schema (**V-Store**). Figure 2a gives an example, where the value of each point contains two parts:

(1) *Spatio-temporal properties*, which consists of the latitude lat, longitude lng, and time t of this GPS point.

(2) *Other properties*, which includes the moving object identifier oid, the trajectory id tid, and other property readings.

V-Store regards each GPS point as an independent entity, which lacks full information of a trajectory. Hence, it is not fit for trajectory queries, especially for similar query and k-NN query. Besides, the number of rows is equal to the number of GPS points, which results in prohibitively numerous key-value entries. More key-value entries need more disk storage space. When retrieving the same number of trajectories, it triggers more disk I/Os, which further hurts the query efficiency.

Horizontal Storage Schema. To address the aforementioned issues, this paper proposes a novel horizontal storage schema, i.e., **H-Store**, to store each trajectory in a single row. As shown in Figure 2b, the value of each entry contains four parts:

(1) *Spatio-temporal properties*, which includes the MBR mbr, the start and end time t_s and t_e, and the start and end positions p_s and p_e of a trajectory.

(2) *GPS point list*. The GPS points in a trajectory are first serialized using Kryo, and then compressed with GZip. It not only reduces the storage cost tremendously, but also improves the efficiency of storing and querying by reducing disk I/Os.

(3) *Signature*. In most scenarios, a trajectory only locates in a very small part of its MBR, i.e., the MBR of a trajectory cannot represent its position exactly. To this end, we creatively design a signature, which provides finer-grained information of the trajectory location. As shown in Figure 2c, the MBR of a trajectory is divided into $\alpha \times \alpha$ disjoint regions with equal size, and each region is numbered. The signature is a binary sequence of $\alpha \times \alpha$ bits. If one or more GPS points of the trajectory are located in a region, the corresponding bit is set to 1, otherwise set to 0. A bigger α means a finer representation, but it requires more storage space and query time. In our implementation, we set $\alpha = 4$.

(4) *Other properties*. Like V-Store, we also store the moving object id oid, trajectory id tid, and other related properties.

In the following, we will elaborate the design of keys for each indexing table using H-Store in TrajMesa.

B. ID Temporal Indexing

To support ID temporal query efficiently, TrajMesa stores a copy of trajectories using `attribute indexing strategy` of GeoMesa. Figure 3 shows the key combination of ID indexing table, where $shard$ is a random number to achieve load balance, oid is the moving object ID, XZT is transformed from the trajectory time span (we will detail it later), and tid is the trajectory ID.

$$shard + oid + \overbrace{BinNum(2\text{ bytes}) + ElementCode(8\text{ bytes})}^{XZT} + tid$$

Fig. 3. Key Composition of ID Indexing Table.

We propose XZT to convert time ranges into one-dimensional keys. As there is no limit for the time dimension, we divide the time line into disjoint time period bins (one day, one week, one month, or one year). For each bin, we encode the time ranges whose start time locates in it. Specially, as shown in Figure 3, XZT consists of two parts:

(1) *Bin Num*. It indicates which time period bin that a time

range belongs to, which is defined by Equ (2).

$$Bin(t_s) = \lfloor (t_s - RefTime) \div BinLen \rfloor \tag{2}$$

where t_s is the start time of a time range, $RefTime$ is the reference time (e.g., 1970-01-01T00:00:00Z), and $BinLen$ is the number of seconds in a bin.

(2) *Element Code*. It represents the offset of a time range in its time bin. There are three steps to get the element code:

- *Time Range Transformation*. It transforms the time range $[t_s, t_e]$ into an offset time range $[t'_s, t'_e]$ according to Equ (3). Note t'_e is also calculated based on the bin number of t_s.

$$Trans(t, Bin) = t - Bin \times BinLen - RefTime \tag{3}$$

- *Sequence Calculation*. This step gets a binary sequence S in a way similar to binary search. As shown in Figure 4a, we recursively find a line segment $L = [tl_s, tl_e]$ in the time period bin to represent the time range $[t'_s, t'_e]$. If the start time t'_s is located in the left half part of the search space, we append 0 to S; otherwise, we append 1. This procedure is terminated when at least one of the following conditions is not met.

$$\begin{cases} tl_s \le t'_s \wedge tl_e + tl_e - tl_s \ge t'_e & (I) \\ |S| < g & (II) \end{cases} \tag{4}$$

Condition (I) guarantees that $[tl_s, tl_e + tl_e - tl_s]$ (we call it the extended time range of $[tl_s, tl_e]$, by double its time span) contains the time range $[t'_s, t'_e]$. Condition (II) means that the length of S is not greater than a user specified constant g. As shown in Figure 4a, line segment $[t/4, t/2 + t/2 - t/4]$ contains $[t'_s, t'_e]$, but $[t/4, 3t/8 + 3t/8 - t/4]$ does not. Hence, the sequence of $[t'_s, t'_e]$ is 01.

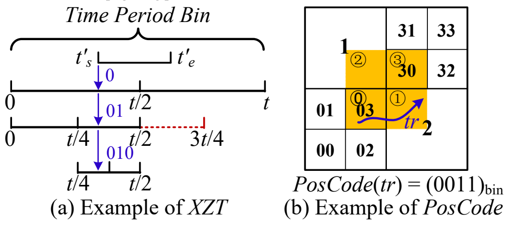


Fig. 4. Example of Indexing.

- *Code Generation*. This step generates a long integer code from the binary sequence, according to Equ (5). It can be regarded as a process of converting binary to decimal.

$$C(S) = \sum_{i=0}^{l-1} s_i \times (2^{g-i} - 1) + 1 \tag{5}$$

C. Spatial Range Indexing

We build spatial range indexing table for spatial range query, similarity query and k-NN query based on `XZ2 indexing strategy` of GeoMesa. Figure 5 shows the key composition of spatial range indexing table, where *shard* is a random number, XZ^+ is transformed from the spatial information of a trajectory (detailed later), and *tid* is the trajectory ID.

$$shard + \overbrace{PosCode(4 \text{ bits}) + XZ2(8 \text{ bytes})}^{XZ2^+} + tid$$

Fig. 5. Key Composition of Spatial Range Indexing Table.

$XZ2$ [14] is extended from $Z2$ [15], which finds a region to represent non-point data (e.g., lines). As shown in Figure 4b, tr is transformed to 03, as the yellow region R (extended from 03 by doubling its width and height) JUST covers tr. However, R is too big for tr, as tr only crosses a small portion of it. In view of that, we propose $PosCode$ to indicate the position of trajectories more accurately. We divide the yellow region R into $\beta \times \beta$ disjoint equal-sized areas, and number them. Then $PosCode$ is a sequence of $\beta \times \beta$ bits. If at least one GPS point of the trajectory locates in an area, the corresponding bit is set 1, otherwise set 0. We set $\beta = 2$ in implementation.

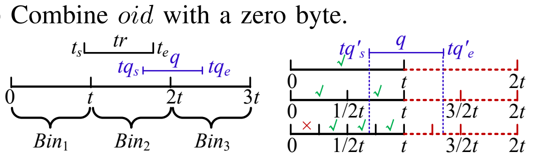
IV. Query Processing

A. ID Temporal Query

Query Window Generation. The key of ID temporal indexing table is $shard + oid + BinNum + ElementCode + tid$. Given an ID temporal query q with a time range $[tq_s, tq_e]$ and a moving object ID oid, we get query windows by:

(1) Enumerate all possible values of $shard$.

(2) Combine oid with a zero byte.



(a) Time Period Bin Selection (b) Offset Key Range Generation

Fig. 6. Example of Query Window Generation.

(3) $BinNum$ generation, which finds a list of time period bins whose extended time range is overlapped with $[tq_s, tq_e]$. Suppose $b_m = Bin(tq_s)$ and $b_n = Bin(tq_e)$, then the bins b_i, $m - 1 \le i \le n$, are selected.

(4) Offset key range generation. For each b_i, we generate a list of offset key ranges. We first get the offset time range $[tq'_s, tq'_e]$ according to Equ (3) based on b_i, then recursively check the extended time range $Xcur$ of a time range cur until the max level g is reached. If $Xcur$ is contained in $[tq'_s, tq'_e]$, we add a key range that represents all sub time ranges in cur. If $Xcur$ is overlapped with $[tq'_s, tq'_e]$, we add a key range that exactly stands for cur to result. Finally, we merge key ranges if a key range is adjacent to its successor, to reduce the number of key ranges. Figure 6b gives an example of offset key range generation, where $g = 2$. The qualified time ranges are checked. Furthermore, the key ranges of $[t/4, t/2]$, $[t/2, 3t/4]$, $[3/4t, t]$, and $[1/2t, t]$ can be merged into one.

(5) Query window combination. We combine $shard$, oid, $BinNum$, and key ranges into query windows.

Query Execution. TrajMesa triggers `Scan` operations over the underlying data store in parallel using query windows.

Result Refinement. When all `Scan` operations are finished, we removes unqualified trajectories.

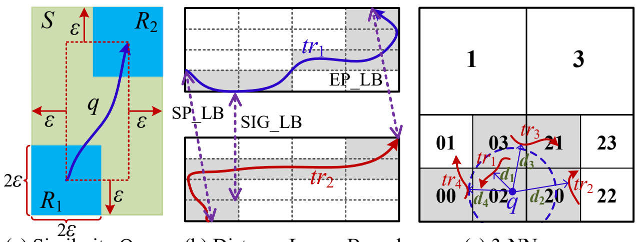
B. Spatial Range Query

Query Window Generation. Spatial range query is based on spatial range indexing table, whose keys follow a pattern of $shard + PosCode + XZ2 + tid$. There are four substeps:

(1) $shard$ generation, the same with ID temporal query.

(2) Spatial key range generation. This step generates a list of key ranges by the given spatial range query q, which is similar to ID temporal query but extended to two dimensions [14].

(3) $PosCode$ generation. As discussed in Section III-C, we divide an extended area into $\beta \times \beta$ disjoint areas, which can be represented by $\beta \times \beta$ bits B. For each extended area of a



(a) Similarity Query (b) Distance Lower Bound (c) 3-NN query

Fig. 7. Example of Trajectory Queries.

qualified e, if q intersects with the i-th area, then $B[i] = 1$, otherwise $B[i] = 0$. A $PosCode$ of e must satisfy Equ (6):

$$PosCode(e)\&B \neq 0 \tag{6}$$

where $(*\&*)$ means bitwise AND operation. For each qualified e, we generate all satisfied $PosCode$s, whose number is between $2^{\beta\times\beta-1}$ (if q intersects with only one area) and $2^{\beta\times\beta}-1$ (if q intersects with all areas).

(4) Query window combination, which combines $shard$, key ranges, and $PosCode$s into query windows.

Query Execution. Similar to ID temporal query, but we use the spatial range indexing table in this case.

Result Refinement. Similar to ID temporal query, but we refine trajectories by the given spatial range.

C. Similarity Query

Similarity query regards spatial range query as a building block. As shown in Figure 7a, given a query trajectory q with a distance threshold ε (we would transform it from km to coordinate degree), we get two spatial ranges $R_1 = \{lat_1-\varepsilon, lng_1-\varepsilon, lat_1+\varepsilon, lng_1+\varepsilon\}$ and $R_2 = \{lat_n-\varepsilon, lng_n-\varepsilon, lat_n+\varepsilon, lng_n+\varepsilon\}$, where (lat_1, lng_1) and (lat_n, lng_n) are the start and end points of q, respectively. All similar trajectories should be contained in the spatial range query result $\mathcal{T}' = SR\_query(\mathcal{T}, R_1) \cap SR\_query(\mathcal{T}, R_2)$.

Lemma 1. *All similar trajectories of q are contained in $\mathcal{T}'$, in terms both of f_F and f_H.*

As the complexities of f_F and f_H are both $\mathcal{O}(|tr|\times|q|)$, we propose 3 types of pruning strategies (as shown in Figure 7), all of which can be calculated in a constant time complexity.

(1) *MBR Pruning*. Suppose $q.mbr = \{lat_{min}, lng_{min}, lat_{max}, lng_{max}\}$, we get $S = \{lat_{min}-\varepsilon, lng_{min}-\varepsilon, lat_{max}+\varepsilon, lng_{max}+\varepsilon\}$. The MBRs of all similar trajectories should be full contained in S.

(2) *SEP_LB*. We propose a lower bound for Fréchet distance based on the start and end points of two trajectories.

$$SEP\_LB_{f_F}(q, tr) = \max\{d(q.p_s, tr.p_s), d(q.p_e, tr.p_e)\} \tag{7}$$

Lemma 2. *If $SEP\_LB_{f_F}(q, tr) > \varepsilon$, then $f_F(q, tr) > \varepsilon$.*

(3) *SIG_LB*. We propose a signature lower bound based on the trajectory signature.

$$SIG\_LB_{f_F}(q, tr) = SIG\_LB_{f_H}(q, tr) =$$
$$\max\{\max_{r_q\in Sig(q)}\min_{r_{tr}\in Sig(tr)} d(r_q, r_{tr}), \max_{r_{tr}\in Sig(tr)}\min_{r_q\in Sig(q)} d(r_{tr}, r_q)\} \tag{8}$$

where $r_q \in Sig(q)$ is a signature region of trajectory q, $d(r_q, r_{tr})$ is the region distance between r_q and r_{tr}.

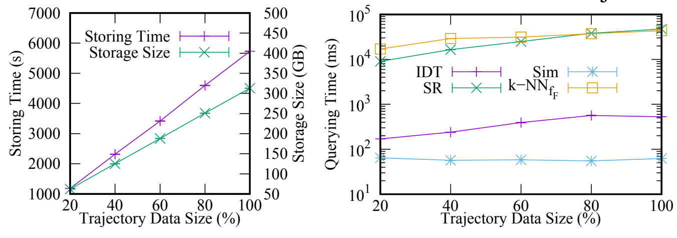
Lemma 3. *If $SIG\_LB_{f_F}(q, tr) > \varepsilon$, then $f_F(q, tr) > \varepsilon$. Beside, If $SIG\_LB_{f_H}(q, tr) > \varepsilon$, then $f_H(q, tr) > \varepsilon$.*

D. k-NN Query

The main idea of k-NN query is to iteratively expand the query spatial range, until the most k similar trajectories are found. Figure 7c gives an example of k-NN point query, we trigger 5 spatial range queries with a spatial range of a certain resolution, following a descending distance order between the query point q and the target region r. Once the distance between q and r is larger than the distance d_k between q and the k-th nearest trajectory tr_k, we can stop the query process, and return the result.

V. Experiments

We conduct a set of experiments using a synthetic trajectory dataset, whose size is over 1360GB. As shown in Figure 8a, when the data size gets bigger from 20% to 100%, both storing time and storage size grow linearly, because more trajectories need to be processed. Storing about 1T data only needs about 1.5 hours and 313GB disk space, which is due to the novel underlying storage schema and compression mechanism. Figure 8b shows that the time of all queries increases with a bigger data size, as more trajectories are qualified, and it triggers more disk I/Os. It is interesting to see that the query time of similarity query is less than that of spatial range query, although we perform two spatial range queries underlying each similarity query. The reason could be that similarity query prunes most trajectories, and there is much less data returned. The transmission bandwidth acts as a bottleneck in TrajMesa.



(a) Storage Performance. (b) Query Performance.

Fig. 8. Performance of TrajMesa.

References

[1] G. Wu, Y. Ding, and et al., “Mining spatio-temporal reachable regions over massive trajectory data,” in *ICDE*. IEEE, 2017, pp. 1283–1294.
[2] T. He, J. Bao, and et al., “Detecting vehicle illegal parking events using sharing bikes’ trajectories,” *SIGKDD*, 2018.
[3] Z. Chen, H. T. Shen, and et al., “Searching trajectories by locations: an efficiency study,” in *SIGMOD*. ACM, 2010, pp. 255–266.
[4] S. Wang, Z. Bao, J. S. Culpepper, Z. Xie, Q. Liu, and X. Qin, “Torch: A search engine for trajectory data.” in *SIGIR*, 2018, pp. 535–544.
[5] D. Xie, F. Li, and J. M. Phillips, “Distributed trajectory similarity search,” *VLDB*, vol. 10, no. 11, pp. 1478–1489, 2017.
[6] Z. Shang, G. Li, and Z. Bao, “Dita: Distributed in-memory trajectory analytics,” in *SIGMOD*. ACM, 2018, pp. 725–740.
[7] “Hbase,” https://hbase.apache.org/, accessed June, 2019.
[8] “Geomesa,” https://www.geomesa.org/, accessed June, 2019.
[9] S. Ruan, R. Li, and et al., “Cloudtp: A cloud-based flexible trajectory preprocessing framework,” in *ICDE*. IEEE, 2018, pp. 1601–1604.
[10] H. Alt and M. Godau, “Computing the fréchet distance between two polygonal curves,” *IJCGA*, vol. 5, no. 01n02, pp. 75–91, 1995.
[11] S. Nutanong, E. H. Jacox, and H. Samet, “An incremental hausdorff distance calculation algorithm,” *VLDB*, vol. 4, no. 8, pp. 506–517, 2011.
[12] Y. Zheng, “Trajectory data mining: an overview,” *TIST*, vol. 6, no. 3, p. 29, 2015.
[13] J. Bao, R. Li, X. Yi, and Y. Zheng, “Managing massive trajectories on the cloud,” in *SIGSPATIAL*. ACM, 2016, p. 41.
[14] C. B, G. K, and H.-P. K, “Xz-ordering: A space-filling curve for objects with spatial extension,” in *SSD*. Springer, 1999, pp. 75–90.
[15] H. S, *Space-filling curves*. Springer Science & Business Media, 2012.