A Framework to Support Continuous Range Queries over Multi-Attribute Trajectories

Jianqiu Xu, Zhifeng Bao, Hua Lu, *Senior Member, IEEE*

Abstract—Emerging applications over spatio-temporal trajectories require representing the data from diverse aspects. We study multi-attribute trajectories each of which consists of a sequence of time-stamped locations and a set of attributes characterizing diverse aspects. We investigate continuous range queries over multi-attribute trajectories. Such a query returns trajectories whose attributes contain expected values and whose locations are always within a distance threshold to the query trajectory during the entire overlapping time period. To efficiently answer the query, an optimal method of partitioning the trajectories is proposed and an index structure is developed to support the combined search using both spatio-temporal parameters and attribute values. Query algorithms and auxiliary structures are developed, accompanied with optimization strategies and thorough theoretical analysis. Using both real and synthetic datasets, we carry out comprehensive experiments in a prototype database system to evaluate the efficiency and scalability of our designs. The experimental results show that our approach outperforms six alternative approaches by a factor of 5-50x on large datasets.

Index Terms—multi-attribute trajectories, continuous range, index structure, approximate computation

1 Introduction

The increasing prevalence of GPS-equipped mobile devices has led to an explosion of *spatio-temporal trajectories*. In the last decade, a rich body of research has been conducted on processing such data [17] [5] [29] [16] [27]. Emerging applications perform data analytics and query processing over big trajectories to enhance their services. Due to COVID-19 virus pandemic recently, the system requires to find out people who have been close to infected or likely infected persons [13] [14]. To achieve this task, one needs not only time-stamped locations but also an attribute describing the state of the person: {Safe, Infected, Likely infected}. The system utilizes the attribute to determine from when and where an infected person will further infect other persons rather than simply reporting two trajectoires which have been close to each other for a while. This enhances the searching efficiency and effectiveness as a large search space will be involved if only spatio-temporal trajectories are analyzed and the result may not be accurate. The increasing popularity of car-calling and ride-sharing services (e.g., DiDi and Uber) have lead to a large number of driver and passenger trajectories. To enhance the schedule capability, an important task is to analyze locations of passengers and drivers at which passengers request pick-up services and drivers who are not in service. The overall pick-up time can be reduced if passengers would like to move a short distance to places at which it takes much less time and effort for drivers to reach than their original places, in particular, at peak hours. The places are those routes at which drivers are not in service and within a short distance to the passenger's current location. We believe that it requires less effort to let passengers move to an appropriate pick-up place than letting drivers move to the passenger's place. Usually, a passenger's location is equivalent to the pick-up request location. However, in many all-time highly crowded places such as train station and airport, passengers can only be picked up in certain areas. In this context, passengers need to move to certain places. Also, during traffic period the time to be picked up would be less if the passenger moves to a nearby place with low congestion. The passenger's current location may be difficult to reach for the driver due to one-way street or making a U turn. We generalize the case by considering a passenger's trajectory rather than using a spatial location.

Consider the following task. Given a driver trajectory who is *free*, report passenger trajectories who have been within the d-distance (e.g., 1 km) to the driver and request pick-up services. To achieve the task, trajectory databases at first require to extend the data representation by integrating descriptive attributes into spatio-temporal trajectories. There are *driver* and *passenger* trajectories, and each trajectory contains the status information such as whether the driver is *free* and the passenger has been picked up. Furthermore, drivers' vehicles have several types such as *taxi*, *tailored taxi* and *express* which have different charges and need to be defined as well. We call spatio-temporal trajectories associated with descriptive

- *Jianqiu Xu is with the Department of Computer Engineering and Science, Nanjing University of Aeronautics and Astronautics, China. E-mail: jianqiu@nuaa.edu.cn*
- *Zhifeng Bao is with the School of Computing Technologies, RMIT University, Australia. E-mail: zhifeng.bao@rmit.edu.au*
- *Hua Lu is with the Department of People and Technology, Roskilde University, Denmark. E-mail: luhua@ruc.dk*

attributes *multi-attribute trajectories*. The query is called *Continuous Range query with Attributes*, CRA for short.

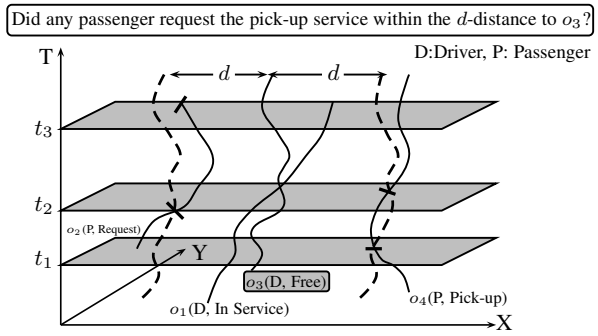


Fig. 1. Example of querying multi-attribute trajectories

The CRA reports trajectories satisfying the criteria: (i) *attribute consistency* and (ii) *time-dependent distance constraint*. Each trajectory is associated with a number of attribute values and only those containing the query value will be further evaluated. In Figure 1, although o_1 (DRIVER, In Service) is within the d-distance to the query target o_3, it does not fulfill the attribute condition. The query defines a dynamic searching area as the driver's location changes over time. This complicates the evaluation as trajectories may be within the distance for a while and then not. Trajectories are decomposed and only pieces of movements within the query distance are considered. In the example, o_2 is within the range during $[t_2, t_3]$, but it does not satisfy the condition during $[t_1, t_2]$. As a result, only the movement at $[t_2, t_3]$ is reported. This differs from traditional range queries in trajectory databases [22] [3] [33] in which the query region is static. This leads to different results. Specifically, continuous range query reports trajectories at each piece of the query time interval because distances between trajectories change over time and only pieces of movements falling in the range will be returned. In contrast, the traditional range query considers a spatial range for a time interval and the distance evaluation is conducted between a line (projecting trajectories into the 2-D space) and an rectangle (or circle). Nearest neighbor queries [9] [26] report the closest trajectory to the target but there is no distance constraint. Consequently, the distance could be very large in practice and the results are not the same as those of our query.

Recently, trajectories featuring multiple attributes have received increasing attention [30] [34] [4] [25] [23]. Such data opens door to understand trajectories along different dimensions simultaneously. What distinguish multi-attribute trajectories from them are as follows: (i) **Semantics and data representation.** Spatial and spatio-temporal trajectories are enriched by keywords and labels for describing individual locations, whereas multiple attributes can be location-dependent or location-independent. Semantic trajectories do not attach labels to the overall movements and usually semantic locations are sparsely defined as only a few locations of trajectories have keywords and labels, e.g., POIs. Our attributes are assigned to the overall trajectory, otherwise redundant data are stored. (ii) **Processed queries.** Queries in semantic trajectories incorporate the measurement of spatial and textual relevances in order to find the most relevant trajectories, e.g., *ranked retrieval* and *top-k retrieval*. The returned trajectories typically fulfill the condition at a certain time point. Consider the query "*return top-2 trajectories that pass Costa and Pizza Hut in the city center between [10am, 12am]*". The procedure will evaluate trajectories containing "*coffee*" and "*pizza*" and order them based on their distances that combine spatial closeness and text relevance. However, coffee and pizza are related to specific locations and thus cannot be attached to the overall movement. The evaluation is performed on certain time points instead of all time points during an interval.

Efficient management of multi-attribute trajectories requires underlying systems to be complemented in terms of data representation and indexing methods. This motivates us to at first model attributes and integrate them with spatio-temporal trajectories into a unified framework. We primarily focus on processing static attributes (i.e., values do not change over time) and dynamic attributes with low updating frequency. If only static attributes are considered, each trajectory is associated with certain attribute values without updating. If there are dynamic attribute values but the updating frequency is not high, e.g., taxi status, one can partition trajectories with a dynamic attribute into a sequence of sub-trajectories each of which is for the movement containing only one attribute value. If the updating frequency is high, the partition method can be still applied but an approximate representation will be employed. Such a method can also be utilized for attributes with large domains.

Next, an effective and efficient index structure is essentially required as this plays a pivotal role in query processing. It is noteworthy that the well-established spatio-temporal indexes is suboptimal for multi-attribute trajectories because they do not manage attributes and therefore one can not prune the search space for attributes. Consequently, trajectories after the spatio-temporal evaluation are sequentially processed. If the query trajectory has a short lifespan or the distance threshold is small, a few trajectories will be returned. In such a case, the approach still achieves good performance. However, if the spatio-temporal predicate has bad selectivity, sequentially evaluating the attribute predicate for each data trajectory significantly inhibits the performance. Alternatively, one can build an attribute index to at first retrieve trajectories with qualified query attributes. The major drawback of this *attribute-first-pruning* strategy is that trajectories after the evaluation will be processed by either performing the sequential scan or accessing an index built on-the-fly. If the attribute predicate is selective, the sequential scan is acceptable. Otherwise, a large number of trajectories are fetched as intermediate results. Both sequential scan and on-the-fly index building incur high computation costs. Creating an index for each query causes storage overhead, especially when a lot of queries are issued. Consequently, this method is limited in scope.

Most cases, however, prefer a joint index that supports

the combined search on spatio-temporal parameters and attributes. As such, we adapt a standard 3-D R-tree by deploying an optimal partitioning of spatio-temporal trajectories. The goal is to normalize trajectories to create an R-tree with a good shape. An attribute structure is created on top of the R-tree to maintain attribute values. We design a flexible method such that the attribute structure can be discarded if only spatio-temporal trajectories are processed. In this case, the structure is merely a 3-D R-tree, avoiding specific indexes. This provides a general solution for both multi-attribute trajectories and spatio-temporal trajectories rather than developing two systems that process them separately.

Our contributions are summarized as follows: first, we formulate multi-attribute trajectories and continuous range queries over them. Second, an optimal data partition method over trajectories with thorough analysis is proposed. Next, we develop a hybrid index supporting updating as well as efficient algorithms to answer the query. Furthermore, an efficient approximate distance computation method is developed to speed up the evaluation procedure accompanied with space and time complexities analysis. Finally, the proposal is fully implemented in a database system SECONDO. A thorough experimental study is performed using real and synthetic datasets. The results demonstrate that our approach outperforms six alternatives by a factor of 5-50x on large datasets.

The rest of the paper is organized as follows. In Section 2, we review the related work. The studied problem is defined in Section 3. The index structure and query algorithms are proposed in Sections 4 and 5, respectively. Approximate distance computation is presented in Section 6. We perform the evaluation in Section 7, followed by conclusions in Section 8.

2 Related work

There is a substantial body of literature on querying and analyzing spatio-temporal trajectories, e.g., range queries, nearest neighbor queries [9] [6] [26], similar trajectory queries [29], trip prediction [20] and route planning [21]. In order to comprehensively understand mobility data, extensive information is essentially required in addition to time-stamped locations [36]. Extracting semantics from spatio-temporal trajectories is investigated by identifying stops or moves and annotating relevant locations with semantics such as *hotel* and *restaurant* [30]. The *partition-and-summarization* approach automatically generates texts to highlight semantic behavior for spatio-temporal trajectories [18]. Then, one forms a sequence of time-stamped locations with semantic labels, called *semantic trajectories*. Attaching semantic labels to locations enables users to perform queries and analytics considering semantic interests and location preferences. Existing queries fall into two categories: (i) *Ranking* and *top-k*. Relevant queries consider actions and activities that users can take at particular places such as *sport* and *dining*. A *conjunctive* query returns k trajectories whose semantics contain the query and have the shortest minimum match distance [35]. An *approximate keyword search* retrieves trajectories containing the relevant query keywords and having short travel distance [34]. A *top-k exemplar trajectory* query [25] consists of a set of locations with keywords and aims to find the most relevant trajectories in terms of the spatial and textual similarity. (ii) *Data mining and analytics*. Frequent sequential patterns can be found to reflect movement regularity by considering spatial compactness, semantic consistency and temporal continuity simultaneously [32]. A regional semantic trajectory pattern mining problem is studied in [4], the aim of which is to identify all the regional sequential patterns in semantic trajectories including global and local frequent patterns. A detailed discussion on semantic trajectories can be found elsewhere [15] [31]. Semantic trajectories focus on location-dependent data and mainly target ranking queries that combine the spatial proximity and textual similarity. In contrast, multi-attribute trajectories support both location-dependent and location-independent information, leading to a general data representation. We deal with continuous queries that report trajectories containing the query attribute value and falling in a dynamic area.

A systematic study is performed to capture a wide range of meanings related to locations including street names, transportation modes and speed profile [10]. A time-dependent label is defined to represent the so-called *symbolic trajectories*, but time-dependent locations are not included in the model. Later, a framework of analyzing large sets of movement data having time-dependent attributes is developed [24] [23]. The work is based on symbolic trajectories and includes spatio-temporal trajectories in the data representation. They aim to support pattern matching queries on tuples of time-dependent values. A new pattern language is proposed and the superiority is thoroughly analyzed in terms of flexibility and expressiveness. Their works are orthogonal to our work. First, different queries are evaluated. Our attribute and spatio-temporal parameters can be individually evaluated, while they deal with static range queries. Second, their main contribution is a flexible and expressive pattern language and the scalability and performance is not extensively evaluated in terms of the number of attributes and the domain size. Also, the number of trajectories in the evaluation is not large (162,000 spatio-temporal trajectories).

Recently, traditional spatio-temporal indexes have been studied to incorporate semantic information. A hierarchical aggregate grid index called *HAGI* is developed to support heterogeneous kNN queries [19]. The method can be adapted to answer our queries, but it is limited as only one attribute is considered. A function is defined to combine the cost of distances and location-independent attributes, and the query returns trajectories having the k-th smallest function value. Each node in *HAGI* maintains *min* and *max* attribute values of all trajectories stored in the subtree. Although *min* and *max* values may work well for one attribute, they fail to guarantee good pruning ability for multiple attributes as *min* and *max* values are likely from different attributes. Also, the query evaluates trajectories based on a ranking function, whereas we

require the exact match on attributes. Furthermore, the distance in the function is static, whereas we deal with time-dependent distances.

To answer spatial keyword range queries on trajectories, a hybrid index called IOC-Tree is proposed [12]. The structure consists of an inverted index and a set of 3-D quadtrees termed octrees. Attributes are defined as keywords. The inverted index is responsible for attribute values, each of which is associated with an octree storing relevant trajectory points. However, for multi-attribute trajectories, each octree will contain all location points of the trajectory. A grid index is established to organize spatio-temporal trajectories with activities in a hierarchical manner [35]. A similar structure is developed to incorporate both spatial and semantic information for approximate keyword search [34]. The grid is a spatial index that is extended to maintain trajectories based on the spatial and activity proximity for ranking queries. This line of work is not applicable to our problem. On the one hand, our attributes are not related to locations and therefore it does not make sense to group trajectories by considering both spatio-temporal locations and attributes. On the other hand, our query reports trajectories rather than individual locations.

3 Problem Definition

Let $\mathcal{O}$ be a set of multi-attribute trajectories. Each $o \in \mathcal{O}$ is denoted by o(*Trip*, *Att*) in which o.*Trip* and o.*Att* refer to a spatio-temporal trajectory and attributes, respectively. A spatio-temporal trajectory is typically defined by a data type *mpoint* [11]. Table 1 gives the representation of multi-attribute trajectories.

TABLE 1
Representing multi-attribute trajectories

Id: *int*	*Trip*: *mpoint*	*Att*: *att*
o_1	location+time	(Driver, In Service)
o_2	location+time	(Passenger, Request)
o_3	location+time	(Driver, Free)
o_4	location+time	(Passenger, Pick-up)

We model descriptive information by multiple attributes. Let A be the set of multiple attributes. The ith attribute and its domain are denoted by $A[i]$ and *dom*$(A[i])$ ($i \in 1,..., |A|$), respectively. We assume that each *dom*$(A[i])$ is represented by a set of positive integers and define a data type $D_{\underline{\text{att}}}$ for the set of attributes. For readability, we use symbols to denote attribute values.

Definition 1 ***Multi-attribute***

$D_{\underline{att}} = \{(a_1, ..., a_{|A|}) | \ a_i \in dom(A[i]), \ i \in \{1, ..., |A|\}\}$ *such that*

(i) $\forall i \in \{1,..., |A|\}$: $dom(A[i]) \subset N^+$;

(ii) $\forall \ i, \ j \in \{1,..., |A|\}$: $i \neq j \Rightarrow dom(A[i]) \cap dom(A[j]) = \varnothing$.

Attribute semantics depend on real applications. The running example defines *dom*(Type) = {*Driver*, *Passenger*} and *dom*(Status) = {*In Service*, *Free*, *Request*, *Pick-up*}, while other applications may need relevant information such as vehicle type {TRUCK, BUS} and transportation modes {WALK, BICYCLE, BUS} to analyze trajectories of different vehicles and modes. Let $T(o)$ return the time period of a trajectory. We employ the function in [7] to return the time-dependent distance between two trajectories $o_1, o_2 \in \mathcal{O}$, denoted by *dist*$(o_1, o_2, T(o_1) \cap T(o_2))$. Two trajectories are mapped into pieces at the same time interval and the distance is represented by a parabola function; the coefficients depend on locations and velocities.

Definition 2 ***Query attribute***

The query attribute is a tuple defining values for evaluated attributes, denoted by $Q_a = (a_1, ..., a_{|A|})$, $Q_a[j] \in dom(A_j) \cup \{\perp\}$.

The query predicate Q_a defines a component for each attribute. A query may specify one or several attributes. Let $Q_a[j]$ refer to the jth attribute value. We define an operator called **contain**$(o.Att, Q_a)$ that returns *true* if $\forall \ Q_a[j] \neq \perp$: $o.Att[j] = Q_a[j]$. We also support queries with multiple values. To achieve this, elements in Q_a are extended to sets of values, i.e., $Q_a = \{X_1, ..., X_d\}$ in which X_i is a set of attribute values. Accordingly, the operator **contain** is extended: **contain**$(o.Att, Q_a)$ returns *true* if $\forall \ X_i \in Q_a \wedge X_i \neq \emptyset$: $o.Att[j] \in X_i$.

The studied query CRA is formulated below.

Definition 3 ***Continuous Range queries with Attributes***

Given a query trajectory o_q*, a threshold* d *and an attribute predicate* Q_a*, CRA returns* $\mathcal{O}' \subseteq \mathcal{O}$ *such that* $\forall \ o' \in \mathcal{O}'$: (i) ***contain***$(o'.Att, Q_a)$; $(ii) \exists \ \Delta T = T(o_q) \cap T(o')$: $\forall \ t \in \Delta T$, *dist*$(o_q, o', t) \leq d$.

There are two variations: (i) The location of o_q does not change over time such that the query returns trajectories whose distances are smaller than d to a spatial point. (ii) The query returns a trajectory as long as there is an instant at which the distance between the data trajectory and the query trajectory is smaller than d, i.e., $\exists \ \Delta T = T(o_q) \cap T(o')$: $\exists \ t \in \Delta T$, *dist*$(o_q, o', t) \leq d$. Table 2 lists the notations frequently used in the paper.

TABLE 2
Notations

Notation	Description
$\mathcal{O}$	the multi-attribute trajectory database
o, $T(o)$	a multi-attribute trajectory and its time period
$\|A\|$	the number of attributes
dom$(A[i])$, *dom*(A)	the domain $A[i]$ and the overall domain
o_q, d	query trajectory and query distance
Q_a	query attribute
t	a time point
δ	grid granularity
f	fanout of R-tree node

4 The Index Structure

We design an index structure named *GR*^{2}*-tree* including two components: GR-tree and $\underline{\text{R}}_{\text{att}}$. The GR-tree is an adapted 3-D R-tree built on spatio-temporal trajectories and R_{att} is a relation for managing attribute values.

4.1 GR-tree

4.1.1 Partitioning spatio-temporal trajectories

Trajectories have different distributions over time and space. We would like to decompose them into pieces having similar sizes in terms of spatial and temporal dimensions. This will benefit the index structure because spatio-temporal extents of nodes are similar, derivations among nodes are small and the area of *dead space* [1] is reduced. The time dimension is partitioned into a set of equal-sized intervals $\{T_1,...,T_K\}$ ($K > 1$) and the 2-D space is partitioned into $\delta \times \delta$ equal-sized cells. Given a multi-attribute trajectory, its spatio-temporal trajectory is split into a set of so-called *cell trajectories*, each of which represents the movement within a cell during an interval $T_k \in \{T_1 ,...,T_K\}$.

Definition 4 ***Cell trajectory***

Let Cell(o, t) return the cell where the trajectory o is located at a time point $t \in T(o)$. A cell trajectory $o[i] \subseteq o$.Trip is part of the overall trajectory that denotes the movement within one cell fulfilling the condition:(i) o.Trip = $\bigcup o[i]$ (ii) $\forall\ t_1, t_2 \in T(o[i])$, Cell$(o[i], t_1)$ = Cell$(o[i], t_2)$; (iii) $\exists\ T_k \in \{T_1,..., T_K\}$, $T(o[i]) \subseteq T_k$.

We partition each $o \in \mathcal{O}$ into a set of cell trajectories. We may encounter the case that one trajectory enters the cell more than once. As a consequence, there are several cell trajectories from the same target corresponding to the same cell. The GR-tree is built on cell trajectories sorted by time, cell id and 3-D bounding box following a bulk loading approach [2].

Example 1. Using the trajectory o_3 in Figure 1, we assume that the 2-D space is partitioned into 4 $\times$ 4 cells and o_3 is contained by one time interval from $\{T_1,...,T_K\}$. The cells intersecting o_3 and o_3's cell trajectories are reported in Figure 2. This is done by (i) determining the set of cells intersecting the 2D bounding box of o_3 and filtering those cells that do not intersect o_3; (ii) decomposing o_3 into cell trajectories each of which is restricted to one cell.

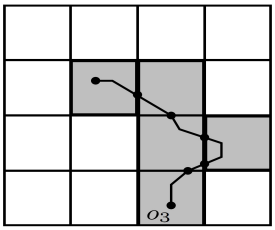
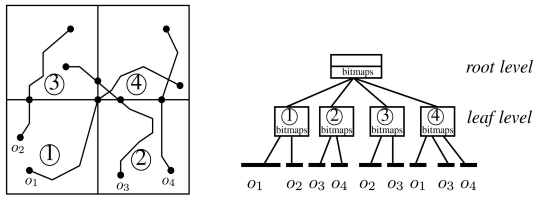


Fig. 2. Partitioning o_3 into cell trajectories

In order to preserve the spatio-temporal proximity, we define that each leaf node only maintains cell trajectories having the same time interval T_i and cell id. Each GR-tree node is supplemented by a bitmap representing the cells intersecting the 2-D bounding box of the node. An *adaptive* mapping between the cells and the bitmap is performed by considering the trajectory distribution among cells. This is motivated by the observation that *dense* cells exhibit higher probability to be accessed than *sparse* cells. More bits are allocated for dense cells and the size of the bit array is set according to the ratio of the number of dense cells to the total number of cells.

Example 2. Using example trajectories in Figure 1, we show the created GR-tree in Figure 3, assuming that trajectories $\{o_1, o_2, o_3, o_4\}$ have the same time interval and the grid granularity is δ = 2. Leaf nodes are marked by their cells. Each node is associated with a bitmap structure for identifying entries containing attribute values. This will enhance the query performance as one can access those entries without performing a linear search.



(a) Cell trajectories (b) GR-tree

Fig. 3. Cell trajectories and GR-tree architecture

4.1.2 Grid granularity

Grid granularity plays a pivotal role in the index design as an arbitrary value cannot guarantee an optimal query performance. If we set a coarse granularity, e.g., δ = 1, all trajectories are located in one cell. Since we put cell trajectories having the same cell id into one leaf node, trajectories will have large extent in x and y dimensions. The created index does not exhibit the spatio-temporal proximity, increasing false positives in query processing.

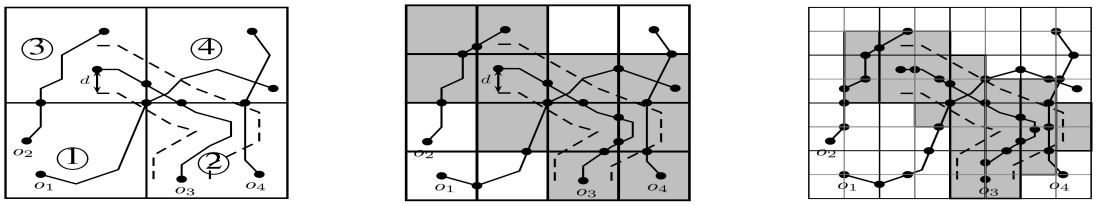
At the opposite end, a fine granularity leads to small cells and each cell contains fewer trajectories having small extent in x and y dimensions. This is good for preserving locality. However, the finer the granularity is (i.e., δ becomes larger), the more GR-tree nodes there are. This is because each cell corresponds to at least one leaf node (a node will overflow if the trajectory number exceeds the node capacity). Some leaf nodes may only contain a few data. Also, the number of cell trajectories grows proportionally as a spatio-temporal trajectory is distributed into all intersecting cells.

Suppose that the 2-D space is a unit space and the side length of a cell is $\frac{1}{\delta}$. Let $X(o) \in$ [0, 1] and $Y(o) \in$ [0, 1] be the length of a trajectory in x and y dimensions, respectively. The number of cell trajectories for o is estimated as $\lceil \frac{X(o)}{\frac{1}{\delta}} \rceil \cdot \lceil \frac{Y(o)}{\frac{1}{\delta}} \rceil$. A large δ increases the number of cell trajectories, leading to more storage overhead. Each node access requires one disk I/O and the increasing I/Os deteriorate the query performance.

Example 3. As shown in Figure 4(a), we will visit all cells under the setting δ = 2 because they are within the d-distance to o_3. However, in cells ③ and ④, cell trajectories of o_2 and o_4 do not fulfill the distance condition.

1. The space is contained by the node but there are few or no data. This means that the area will be evaluated but few trajectories are there or even no trajectory exists.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3100650, IEEE Transactions on Knowledge and Data Engineering



(a) δ = 2 (b) δ = 4 (c) δ = 8

Fig. 4. Coarse and fine grid granularities

Alternatively, we can partition the space by setting δ = 4, as illustrated in Figure 4(b). As a consequence, the search space is reduced as some cells are out of the range. Let us consider δ = 8 by referring to Figure 4(c). Although we can greatly reduce the search space (gray area), more cells (GR-tree nodes) are accessed in comparison with δ = 4.

An effective way is to split trajectories and approximate the resulting sub-trajectories by balancing the number of indexed data and the approximation quality. We analytically derive an optimal granularity as follows. Given a granularity δ, at each T_k the average number of leaf nodes for storing cell trajectories intersecting a cell is approximated by

$$n(\delta) = \lceil \frac{|\mathcal{O}| \cdot T_{avg}}{K} \cdot \frac{1}{\delta^2} \cdot \frac{1}{f} \rceil = \lceil \frac{\mathcal{P}}{\delta^2} \rceil,$$
$$\text{where} \quad \mathcal{P} = \frac{|\mathcal{O}| \cdot T_{avg}}{K \cdot f} \quad \text{and} \quad T_{avg} = \lceil \text{AVG}(\frac{T(o)}{1/K}) \rceil \tag{1}$$

Here, $T_{avg} \in \{1,...,K\}$ is the average number of time partitions that a trajectory contains ($T(o)$ is mapped to the unit space) and f is the capacity of a GR-tree leaf node.

Consider the lower and upper bounds of δ. The lower bound is straightforward, i.e., δ = 1, resulting in a large value $n(1) = \lceil \mathcal{P} \rceil$. Regarding the upper bound, theoretically, one can set $\delta = \infty$. The larger δ is, the denser the cell distribution is. Meanwhile, the smaller each cell will be. For each cell, we maintain leaf nodes storing trajectories located inside the cell. As cells become smaller, they may contain fewer or no trajectory. In fact, the maximum value of δ corresponds to the minimum number of cells such that only one leaf node suffices to store all trajectories inside a cell. It makes no sense to have cells without any trajectory inside. As a result, the upper bound is derived by

$$n(\delta) = \lceil \frac{\mathcal{P}}{\delta^2} \rceil \geq 1 \Rightarrow \delta \leq \lceil \sqrt{\mathcal{P}} \rceil \tag{2}$$

Let $N(\delta)$ be the number of leaf nodes under δ. We have

$$N(\delta) = \begin{cases} \delta^2 & \text{if } \delta \geq \lceil \sqrt{\mathcal{P}} \rceil \\ \delta^2 \cdot n(\delta) = \lceil \mathcal{P} \rceil & \text{else } (\delta \in \{2, ... \lceil \sqrt{\mathcal{P}} \rceil - 1\}) \end{cases} \tag{3}$$

The task is to find an optimal granularity δ such that

$$\delta^* = \text{argmin} N(\delta) \wedge \delta^* = min\{1, 2...., \sqrt{\mathcal{P}}\} \tag{4}$$

We analyze that if $\delta > \lceil \sqrt{\mathcal{P}} \rceil$, $N(\delta)$ exhibits quadratic growth. If $\delta < \sqrt{\mathcal{P}}$, $N(\delta)$ is in fact independent of δ.

Lemma 1 *The optimal granularity*

$$\delta^* = \lceil \sqrt{\mathcal{P}} \rceil \geq 2, \quad \textit{where} \quad \mathcal{P} = \frac{|\mathcal{O}| \cdot T_{avg}}{K \cdot f} \textit{ in which} \tag{5}$$

$|\mathcal{O}|$ is the number of trajectories, f is the node capacity, K is the number of partitions over time and T_{avg} is a value from the set {1,..., K}.

Proof *(i) If $\delta > \delta^*$, we have $N(\delta) = \delta^2 > (\delta^*)^2 = N(\delta^*)$ as the value increases exponentially with δ. (ii) If $\delta < \delta^*$, without loss of generality we have $\delta = \delta^* - 1$.*

$$\begin{aligned} N(\delta) - N(\delta^*) &= (\delta^* - 1)^2 \cdot n(\delta) - (\delta^*)^2 \\ &\geq (\delta^* - 1)^2 \cdot 2 - (\delta^*)^2 \\ &= (\delta^*)^2 - 4 \cdot \delta^* + 2 \\ \textit{let} \quad g(\delta^*) &= (\delta^*)^2 - 4 \cdot \delta^* + 2 = (\delta^* - 2)^2 - 2 \end{aligned}$$

$g(\delta^*)$ is a monotonic increasing function when $\delta^* \geq 2$ and $g(\delta^*) > 0$ for all $\delta^* > 3$.

$$\delta^* = \lceil \sqrt{\mathcal{P}} \rceil > 3 \Rightarrow \frac{|\mathcal{O}| \cdot T_{avg}}{K \cdot f} > 9 \Rightarrow \frac{|\mathcal{O}| \cdot T_{avg}}{K} > 9 \cdot f$$

we have $T_{avg} \in \{1, ..., K\}$

and the condition $N(\delta) > N(\delta^*)$ holds

as long as $\frac{|\mathcal{O}|}{K} > 9 \cdot f$ □

Theoretically, one can set a relatively large K (e.g., K = $|\mathcal{O}|$) such that the condition $\frac{|\mathcal{O}|}{K} > 9 \cdot f$ does not hold. In practice, we can easily achieve the condition by setting an appropriate K for a large number of trajectories as $|\mathcal{O}| \gg 9 \cdot f$. One solution is to choose K such that each time partition is equivalent to the average time interval of all trajectories. We do not make any assumption about data distribution, i.e., trajectories can be uniformly or non-uniformly distributed in space and time.

4.2 The attribute structure

The relation R_{att} records attribute values of multi-attribute trajectories in GR-tree nodes. The attribute values of a leaf node are obtained by accessing the underlying data and the values of a non-leaf node are obtained by performing the union on values of its child nodes. The schema of the relation is R_{att} (*nid*, *a_tr*, *b*).

Each tuple indicates an attribute value contained by a node, in which *nid* is a node id, *a_tr* is a transformed attribute value and *b* is a bitmap. The transformed value is uniquely achieved by interleaving the binary representation of the attribute id and the attribute value. We

use a bitmap with the size l_1 + l_2 in which l_1 bits are for attribute ids and l_2 bits are for attribute values. For example, we have two attributes AD and VEHICLE such that l_1 = 1 is enough, i.e., 0 for AD and 1 for VEHICLE. The attributes AD and VEHICLE have two and three values, respectively. As a result, l_2 = 2 is sufficient. Combining l_1 + l_2 bits can represent all attribute values and each one is unique.

We create a B-tree on R_{att} by combining *nid* and *a_tr* to form the key. The bitmap records the entries of a node containing a particular attribute value, enabling us to only access qualified entries instead of performing a sequential scan. We perform different mapping strategies determined by the size of the bit array $|b|$ and the maximum number of entries in a GR-tree node f (i.e., the *fanout*): (i) $|b| \geq f$, each bit maps to a unique entry. We set $b[i]$ = 1 if the ith $\in$ [0, f) entry contains the value. Otherwise, $b[i]$ = 0. (ii) $|b| < f$, each bit maps to a range of entries and entries for the ith bit are calculated by $[i \cdot \lceil \frac{f}{|b|} \rceil, (i+1) \cdot \lceil \frac{f}{|b|} \rceil]$. We define $b[i]$ = 1 if one of the entries contains the value.

The bitmap index incurs little storage overhead and determines qualified entries by performing the bitwise operation AND. Note that the bitmaps do not define attribute values contained by the node. This method will inhibit the performance due to the limitation of the bitmap size, in particular, when the number of attribute values is large. In contrast, we perform the mapping between the bitmap and entries of a node. The number of entries in a node is limited by f, depending on the page or block size. The number of attribute values may be large for some applications but f is usually not. Thus, we do not need a long bit array for f. Let $|N|$ be the total number of GR-tree nodes. We need $O(|dom(A)| \cdot |N|)$ tuples in R_{att}.

Example 4. We report R_{att} by referring to Figure 5 in which attribute values and bitmaps for the root node N_r are provided. Both original attribute values and transformed values are reported. Let N denote a GR-tree node. Consider Q_a = (Passenger, Request). We use "0" for the attribute "Type" and "01" for the value "Passenger". Combining the two values we have "001". The bitmap in N_r is "1111" as $\{N_1, N_2, N_3, N_4\}$ all contain *Passenger*. The attribute value *Request* is transformed into "110" and the bitmap in N_r is "0101" as N_1 and N_3 contain *Request*.

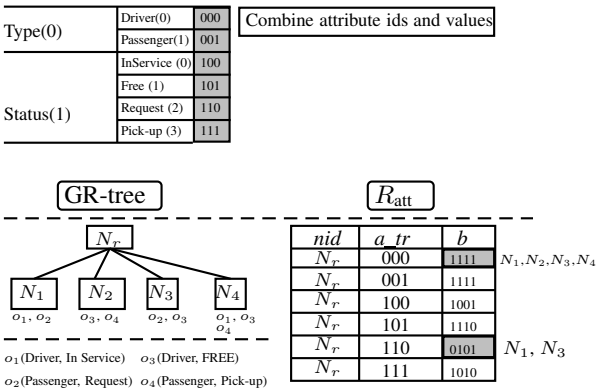


Fig. 5. Example of the attribute relation R_{att} for N_r

4.3 Updating the index

Given a set of incoming multi-attribute trajectories, synchronizing the *GR^2-tree* includes: (i) inserting new arrival trajectories into the GR-tree and (ii) updating the relation R_{att}. Part (i) is achieved by creating a subtree for a group of new trajectories and then finding an appropriate node in the existing GR-tree to locate the subtree. Part (ii) is achieved by inserting new tuples into R_{att} for the new subtree and updating the tuples for attribute values. All nodes on the path from the root node to the node where the subtree is inserted are updated in terms of spatio-temporal data and attributes.

We analyze the complexity of updating the index structure by measuring the number of node accesses. We restrict the height of the subtree to 2 in order to limit the number of incoming trajectories for one time updating. As a result, there will be $O(f^2)$ processed trajectories.

Update complexity. *The cost of updating the index is $O((f+H) \cdot (1 + \frac{dom(A)}{b}))$, in which f is the R-tree node capacity, H is the height of the historical R-tree and b is the block size.*

Proof

Part (i): creating a subtree needs $O(f+1)$ nodes in total and inserting the subtree into the existing index requires accessing $O(H)$ nodes, leading to $O(f+H)$. Part (ii): One needs $O(f \cdot dom(A))$ tuples for storing attribute values in the subtree, leading to $O(\frac{f \cdot dom(A)}{b})$ I/O cost. Updating attribute values for nodes in the historical R-tree requires $O(\frac{H \cdot dom(A)}{b})$ I/O cost. Then, we have $O(\frac{(f+H) \cdot dom(A)}{b})$ I/O cost for part (ii). Combining (i) and (ii), the complexity is $O((f+H) \cdot (1 + \frac{dom(A)}{b}))$. □

5 Query Processing

5.1 An outline

Employing the GR^2-tree, we process the query in three steps, as illustrated in Figure 6.

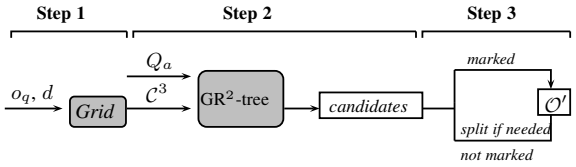


Fig. 6. The query procedure

Step 1 establishes the spatio-temporal area restricted by o_q and d, which is represented by a set of time-dependent cells denoted by $\mathcal{C}^3$. We will present the structure in Section 5.2. Step 2 performs a breadth-first traversal on the GR^2-tree to return a set of candidates, each of which is a cell trajectory that (i) contains Q_a and (ii) has the distance less than d to o_q. The distance is an approximate value calculated by using the minimum bounding boxes of trajectories. A candidate is *marked* if its maximum distance to o_q is less than d. Step 3 iteratively checks the accurate distance between each candidate and the query. If the candidate is *marked*, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3100650, IEEE Transactions on Knowledge and Data Engineering

directly put it into the result set. Otherwise, the actual value is computed. Two trajectories are mapped into pieces at the same time interval. A trajectory may be split because only the piece of movements fulfilling the distance condition is considered. Since step 3 is trivial, we focus on steps 1 and 2 in the following.

5.2 Time-dependent cells

We can quickly determine the cells within the d-distance to the query by utilizing the grid partition. This is achieved by computing the distance between the 2-D bounding box of the query trajectory and the cell. When we traverse the index, the GR-tree nodes that do not intersect the cells can be safely pruned. Usually, a cell is not always within the d-distance to the query as the location of the trajectory changes over time. We report *time-dependent* cells maintained by a composite structure including three components: *cell tree*, *cell set* and *cell list*, denoted by $\mathcal{C}^3$. Let C_i be the set of cells intersecting a cell trajectory $o_q[i]$. Note that $o_q[i]$ is restricted in a cell but we include cells whose borders intersect $o_q[i]$.

• *cell tree*. A binary tree is used to record items of the form $(T(o_q[i]), C_i)$. Each node stores the time interval of $o_q[i]$ and the cells intersecting $o_q[i]$. Items are increasingly sorted on time. The *cell tree* reports all cells within the d-distance to the query during a given time interval.

• *cell set*. The structure stores all cells within the d-distance to o_q and there is no duplicate result, i.e., $\bigcup C_i$.

We define *marked cells* that the cell list maintains.

Definition 5 ***Marked cell***

Let maxdist(c, $o_q[i]$) denote the maximum distance between a cell and a cell trajectory. A cell c is marked if maxdist(c, $o_q[i]$) < d.

• *cell list*. A list of pairs (c, T) is maintained. Each pair contains a *marked* cell and a time interval. The structure determines whether all trajectories in a leaf node are within the d-distance to the query. If positive, the exact distance computation can be avoided as a leaf node stores trajectories whose movements are restricted in the cell. Given a leaf node, if its cell is *marked* and the time is contained by the cell list, all trajectories in the node fulfill the distance condition.

The procedure of constructing $\mathcal{C}^3$ is provided in Algorithm 1. We start by creating the cell tree (lines 2-5). Next, for each node in the tree, we iteratively insert each cell into the cell set and determine whether the cell is marked or not. We insert the marked cell into the list and update the time accordingly.

We now analyze the time complexity of building the structure $\mathcal{C}^3$ for a query trajectory o_q. This depends on two factors: (i) the number of cell trajectories and (ii) the number of cells intersecting each cell trajectory. Part (i) is calculated by

$$\frac{T(o_q)}{1/K} \cdot \frac{X(o_q)}{\frac{1}{\delta}} \cdot \frac{Y(o_q)}{\frac{1}{\delta}} \tag{6}$$

Consider part (ii). Given a cell trajectory $o_q[i]$, we return the cells intersecting $o_q[i]$ the area of which is a rectangle achieved by enlarging $X(o_q)$ and $Y(o_q)$ with $2 \cdot d$. Among all cell trajectories, the maximum number of cells intersecting $o_q[i]$ is calculated by

$$\frac{X(o_q[i])_{max} + 2 \cdot d}{\frac{1}{\delta}} \cdot \frac{Y(o_q[i])_{max} + 2 \cdot d}{\frac{1}{\delta}} \tag{7}$$

Algorithm 1 *TCell*

Input: query trajectory o_q, distance threshold d and grid
Output: time-dependent cells $\mathcal{C}^3$

1: *let Cell(o_q) return cell trajectories of the query*;
2: *Tr* $\leftarrow \varnothing$; ▷ initialize the cell tree
3: **for all** $o[i] \in$ *Cell*(o_q) **do**
4: *search the grid to determine C_i such that* $\forall\ c \in C_i$: *mindist*(c, $o[i]$) $< d$ and c *is marked if maxdist*(c, $o[i]$) $< d$;
5: *insert* $(T(o[i]), C_i)$ *into Tr*;
6: $S \leftarrow \varnothing$, $L \leftarrow \varnothing$; ▷ initialize the cell set and cell list
7: **for all** $(T(o[i]), C_i) \in$ *Tr* **do**
8: **for all** $c \in C_i$ **do**
9: $S \leftarrow c$;
10: **if** c is *marked* **then**
11: **if** $(c, T(o[i])) \notin L$ **then**
12: $L \leftarrow (c, T(o[i]))$;
13: **else**
14: **if** $\exists(c', T') \in L$: $c' = c$ **then**
15: $T' \leftarrow T' \cup T(o[i])$;
16: **return** $\mathcal{C}^3 \leftarrow$ (*Tr*, *S*, *L*);

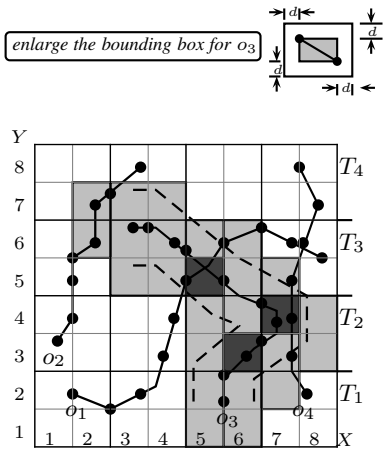
Time complexity. *Given a query trajectory o_q, building the structure $\mathcal{C}^3$ requires $O(C_{o_q} \cdot \log C_{o_q})$ time, in which C_{o_q} is the overall number of processed cells, calculated by*

$$C_{o_q} = \frac{T(o_q)}{1/K} \cdot \frac{X(o_q)}{\frac{1}{\delta}} \cdot \frac{Y(o_q)}{\frac{1}{\delta}} \cdot \frac{X(o_q[i])_{max} + 2 \cdot d}{\frac{1}{\delta}} \cdot \frac{Y(o_q[i])_{max} + 2 \cdot d}{\frac{1}{\delta}}.$$

Proof *The structure $\mathcal{C}^3$ consists of three parts: (i) cell tree, and (ii) cell set and (iii) cell list. Creating (i) needs $O(C_{o_q} \cdot \log C_{o_q})$. Then, for each cell in the tree, we insert it into cell set and cell list, both of which contain $O(C_{o_q})$ cells and each insertion requires $O(\log C_{o_q})$ and $O(1)$ time costs for the cell set and the cell list, respectively. To sum up, we need the time $O(C_{o_q} \cdot \log C_{o_q})$ to construct the structure.* □

Example 5. By referring to Figure 7(a) in which we have K = 4 ($\{T_1, T_2, T_3, T_4\}$) and δ = 8, we enlarge the bounding box of o_3 in both x and y dimensions to find all cells within the d-distance to the query (depicted in gray). Two dashed lines are depicted to help figure out the cells. The time interval $T(o_3)$ intersects $\{T_1, T_2, T_3\}$. The cells $\{c_{5,1}, c_{5,2}, c_{6,1}, c_{6,2}, c_{7,2}\}$ are within the d-distance to the query at T_1, but they should not be considered at T_3. There are three *marked* cells $\{c_{5,5}, c_{6,3}, c_{7,4}\}$ at $T_2 \cup T_3$. Thus, the cell trajectory of o_1 in $c_{5,5}$ and the cell trajectory of o_4 in $c_{7,4}$ can be directly returned

without performing the accurate distance computation (the attribute condition is not considered here). The structure of the time-dependent cells is reported in Figure 7(b). The cell set consists of three parts C_1, C_2 and C_3, partitioned by time intervals. The cell tree is built on cells with corresponding time intervals. Since cells $\{c_{5,5}, c_{6,3}, c_{7,4}\}$ are *marked cells*, they are put into the cell list with time intervals.

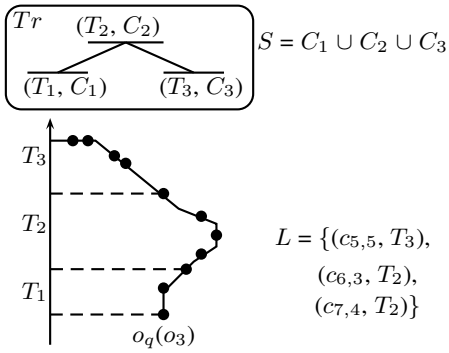


(a) cells within d to o_q and marked cells

$C_1 = \{c_{5,1}, c_{5,2}, c_{6,1}, c_{6,2}, c_{7,2}\}$

$C_2 = \{c_{5,3}, c_{5,4}, c_{6,3}, c_{6,4}, c_{7,3}, c_{7,4}, c_{8,3}, c_{8,4}\}$

$C_3 = \{c_{2,6}, c_{2,7}, c_{3,5}, c_{3,6}, c_{3,7}, c_{4,5}, c_{4,6}, c_{4,7}, c_{5,5}, c_{5,6}, c_{6,5}, c_{7,5}\}$



(b) cell tree, cell set and cell list

Fig. 7. An example of $\mathcal{C}^3$

5.3 Traversing GR^2-tree

This step performs a breadth-first traversal on the index to prune the search space by taking into account both spatio-temporal parameters and attribute values. Given a GR-tree node N, we retrieve its cell bitmap denoted by CBM(N) and determine the cells intersecting the node denoted by *cell*(CBM(N)). The node can be pruned if there is no overlap between *cell*(CBM(N)) and the cell set $\mathcal{C}^3.S$.

Lemma 2 *We can prune a node N if cell(CBM(N)) $\cap$ $\mathcal{C}^3.S = \emptyset$.*

Proof $\forall c \in$ *cell(CBM(N)): $c \notin \mathcal{C}^3.S \Rightarrow$ dist(c, o_q) >d.* □

Let $T(N)$ return the time extent of a node. We utilize the cell tree $\mathcal{C}^3.Tr$ to report all cells during $T(N)$. These cells are within the d-distance to the query o_q. The following pruning strategy is used.

Lemma 3 *Let cell($\mathcal{C}^3.Tr$, $T(N)$) return the cells at $T(N)$ in the cell tree. We can prune N if cell(CBM(N)) $\cap$ cell($\mathcal{C}^3.Tr$, $T(N)$) = $\emptyset$.*

Proof $\forall\ c \in$ *cell(CBM(N)): $c \notin$ cell($\mathcal{C}^3.Tr$, $T(N)$)* $\Rightarrow$ $T(N) \cap T(o_q) = \emptyset \vee$ *dist(c, o_q, $T(N) \cap T(o_q)$) > d.* □

When a leaf node is processed, we iteratively access each cell trajectory in the node to compute the exact distance between each cell trajectory and the query trajectory. This step can be avoided if the following condition holds.

Lemma 4 *All trajectories in a leaf node N fulfill the distance condition if $\exists\ (c', T') \in \mathcal{C}^3.L$: cell(CBM($N$)) = $c' \wedge T(N) \subseteq T'$, where $\mathcal{C}^3.L$ is the cell list.*

Proof *Each item $(c', T') \in \mathcal{C}^3.L$ represents a cell c' such that maxdist(c', o_q) < d at T'. If the leaf node corresponds to such a cell, all trajectories fulfill the condition and the distance computation is omitted.* □

Example 6. Using the example query, we consider $\mathcal{C}^3$(*Tr*, *S*, *L*) and the GR-tree at T_2. We report the structure $\mathcal{C}^3$ at T_2 in Figure 8(a) and illustrate the pruning procedure in Figure 8(b). Starting from N_r, we process nodes level by level. N_a and N_b will be pruned because their cells do not intersect the cell set (Lemma 2). N_c and N_d are further considered. For N_c, we access each child node during which trajectories in the node $c_{6,3}$ are directly reported because the cell exists in the cell list (Lemma 4). For N_d, we open the node and process each child node during which all trajectories in the node $c_{7,4}$ are directly reported (Lemma 4) and the node $c_{8,2}$ is pruned because it is not within the d-distance to o_3 at T_2 (Lemma 3).

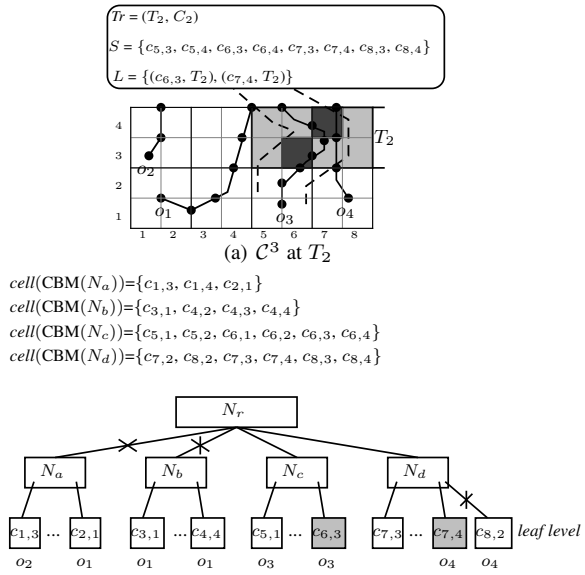
Pruning by attribute values. Given a GR-tree node, we take the node id and the attribute value $a \in Q_a$ to create the key to access R_{att}. The following criterion is used for pruning.

Lemma 5 *Given a node N and an attribute value $a \in Q_a$, let r(nid, a_tr, b) $\in R_{att}$ denote the tuple for N such that a = r.a_tr. We can prune N if* $\bigcap_{a \in Q_a} (r.b) = \emptyset$.

Proof *Let $N[i]$ be a child node. The tuple $r.b$ defines the entries of N containing r.a_tr. We have* $\bigcap_{a \in Q_a} (r.b)$ = $\emptyset \Rightarrow \forall N[i]$: $\nexists\ r_1....,r_{|Q_a|} \in R_{att}$ *such that (r_1.nid = $N[i]$... $r_{|Q_a|}$.nid = $N[i]$) $\wedge$ (r_1.a_tr = $Q_a[1]$...$r_{|Q_a|}$.a_tr = $Q_a[|Q_a|]$).* □

The algorithm of reporting candidates is given in Algorithm 2. Let *cell*($\mathcal{O}$) be the overall cell trajectories. Starting from the root node, we maintain a node list to visit the GR^2-tree level by level. A node is pruned if its cells do not intersect the cells in $\mathcal{C}^3$ or it does not contain Q_a (line 6). For a non-leaf node, we retrieve each

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3100650, IEEE Transactions on Knowledge and Data Engineering



(a) $\mathcal{C}^3$ at T_2

(b) the pruning procedure by GR-tree

Fig. 8. Example of pruning by $\mathcal{C}^3$

qualified child node according to the bitmap (marking entries containing attributes) and put it into the list for further consideration. For a leaf node, we evaluate the approximate distance between the query and each cell trajectory. A cell trajectory is *marked* if its maximum distance to the query is less than d (Lemma 4). Otherwise, we put the trajectory into the candidate set to perform the accurate distance computation later.

Algorithm 2 *AccessGR^2-tree*

Input: o_q, d, Q_a, $\mathcal{C}^3$, *GR^2-tree* and *cell*($\mathcal{O}$)
Output: candidate trajectories

```
1: Cand ← ∅;
2: L ← GR²-tree.Root;
3: while L is not empty do
4:     N ← GetNode(GR²-tree, L.top());
5:     access R_att by N and Q_a;
6:     if N is not pruned by C³ and Q_a then ▷ Lemmas 2, 3 and 5
7:         for all entry in N according to ⋂_{a∈Q_a} (r.b) do
8:             if N is a non-leaf node then
9:                 put the child node into L;
10:            else
11:                get the cell trajectory o ∈ cell(O);
12:                if o.Att contains Q_a then
13:                    if Lemma 4 holds then
14:                        mark o and put it into Cand;
15:                    else
16:                        if mindist(o_q, o, T(o_q) ∩ T(o)) < d then
17:                            Cand ← o;
18: return Cand;
```

5.4 Establishing leaf nodes

We observe that Algorithm 2 traverses the structure in a top-down approach during which non-leaf nodes are accessed at first and then leaf nodes. The returned trajectories are actually stored in leaf nodes. This motivates us to determine the set of leaf nodes containing the result and then directly access leaf nodes without performing the traversal from the root level to leaf level.

Each leaf node only stores trajectories within one cell. Thus, given a cell, all leaf nodes storing trajectories inside the cell can be determined. This is calculated by the 2-D bounding box of a leaf node and the cell. Let f_{cell}: $\mathcal{C} \rightarrow$ *Set(N)* be a function that maps from cells to leaf nodes. Each leaf node is of the format (*nid*, *box*) in which we store the 3-D bounding box of a node. For example, in Figure 9(a) $f_{\text{cell}}(c_{6,3})$ returns the node in which *cell*(o_3) is stored. Given a large number of trajectories, each cell corresponds to a set of leaf nodes because one node may not be sufficient for all trajectories located in the cell.

The structure $\mathcal{C}^3$ contains all cells that are within the d-distance to the target during the query time. Next, we determine the leaf nodes for those cells. Note that the function f_{cell} returns all leaf nodes for a cell but some of them may not intersect the query time and should be pruned. Based on the time-dependent cells returned from Algorithm 1, we provide the method that only accesses leaf nodes to answer the query (see Algorithm 3).

Algorithm 3 *AccessGR^2-treeLeaf*

Input: o_q, d, Q_a, $\mathcal{C}^3$, *GR^2-tree* and *cell*($\mathcal{O}$)
Output: candidate trajectories

```
1: L ← ∅;
2: for all c ∈ C³.S do
3:     N_l ← f_cell(c);
4:     for all n ∈ N_l do
5:         if n is not pruned by Lemmas 2 and 3 then
6:             L ← n;
7: Cand ← ∅;
8: call lines 3-18 in Algorithm 2;
```

Example 7. Based on the time-dependent cells in Figure 8(a), we provide the cell set $\mathcal{C}^3.S$ in Figure 9(a). For each cell, we collect its leaf nodes which are within the d-distance to the target during the query time. In the example, we only access those leaf nodes (depicted in gray) to collect the data without accessing non-leaf nodes $\{N_r, N_a, N_b, N_c, N_d\}$.

6 Approximate Distance Computation

Since computing the exact distance between a cell trajectory and the query trajectory is a costly procedure including mapping pieces of movements into the same time interval, splitting trajectories and determining the time-dependent distance function, an approximate distance is typically used to filter the data that cannot contribute to the result. The distance between two minimum bounding boxes (MBBs) is computed, called *approximate distance*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3100650, IEEE Transactions on Knowledge and Data Engineering

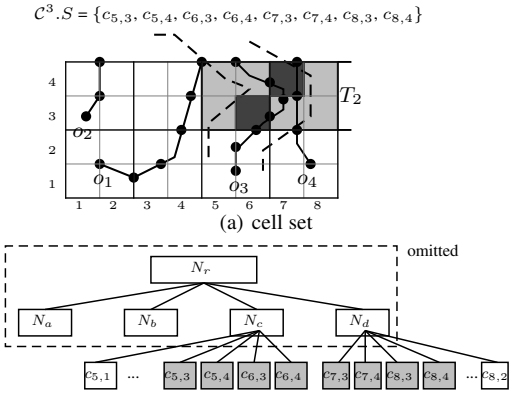


Fig. 9. Example of establishing leaf nodes

computation. To increase the efficiency of establishing the MBB of the query trajectory for a time interval, a common method is to build a bounding box tree (BB-tree) on all MBBs of query trajectory segments [9]. The BB-tree is essentially a binary tree in which each node is associated with a time interval and a rectangle. For a leaf node, the value is taken from the trajectory. For a non-leaf node, the value is the union of its child nodes.

A minimum bounding box is represented by mbb = ($x_{\min}, y_{\min}, x_{\max}, y_{\max}$). Given two MBBs mbb_1 and mbb_2, the union operation that bounds the arguments by MBBs is defined by

$$\begin{aligned}&\text{union}(mbb_1, mbb_2) = \\ &(\text{Min}(mbb_1.x_{\min}, mbb_2.x_{\min}), \text{Min}(mbb_1.y_{\min}, mbb_2.y_{\min}), \\ &\text{Max}(mbb_1.x_{\max}, mbb_2.x_{\max}), \text{Max}(mbb_1.y_{\max}, mbb_2.y_{\max}))\end{aligned}$$

Let m be the number of segments in a query trajectory and C be the total number of candidate cell trajectories involved in the distance computation, respectively.

Time complexity. *Employing the BB-tree, we need* $O(m + C \cdot (\log m + 1))$ *time to perform the computation.*

Analysis. We need $O(m)$ time to build the BB-tree as trajectory segments are already sorted on time. For each cell trajectory, we require the time $O(\log m + 1)$ to traverse the BB-tree and perform the union operation. The time interval of a cell trajectory is usually less than $T(o_q)$. Thus, we need to perform the union operation once to merge two MBBs at different time intervals. As a result, the overall time is $O(m + C \cdot (\log m + 1))$.

To enhance the performance, we propose a method to efficiently retrieve the MBB for a cell trajectory by using a bounding box array, called *BB-array*. This is a two-dimensional array built on-the-fly with the size $B \times B$ ($B \leq m$). First, we partition the query trajectory into B equal-size segments in terms of the time and assign a segment MBB for each *BB-array*[i][i] ($i \in \{0, ..., B - 1\}$). Second, for each row in the array we set *BB-array*[i][j] ($i < j \wedge j \in \{i + 1, ..., B - 1\}$) by performing the union from the ith to jth MBB. That is,

$$\begin{aligned}\textit{BB-array}[i][j] =& \textit{Union}(\textit{BB-array}[i][i], \text{mbb}_{i+1}) \\ \text{mbb}_{i+1} =& \textit{Union}(\textit{BB-array}[i+1][i+1], \text{mbb}_{i+2}), \\ &\ldots \\ \text{mbb}_j =& \textit{Union}(\textit{BB-array}[j-1][j-1], \\ & \textit{BB-array}[B-1][B-1])\end{aligned}$$

Example 8. We report the BB-array built on o_q in Figure 10 by defining $B = 3$.

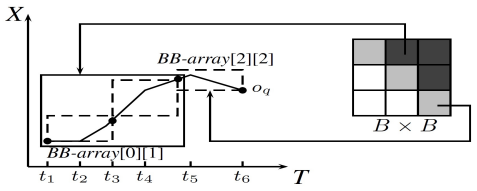


Fig. 10. The *BB-array* built on o_q

Space complexity. *The BB-array's storage cost is* $O(m^2)$.

Given a cell trajectory intersecting the query, we are able to retrieve the MBB at a constant cost by employing the BB-array. The overall time of performing the approximate distance computation is as follows.

Time complexity. *Employing the BB-array, we need* $O(m^2 + C)$ *to process all cell trajectories.*

Analysis. We need $O(m+B^2) = O(m+m^2)$ to create the BB-array as one needs to perform the partition and create the two-dimensional array. For each cell trajectory, its start and end time points correspond to indexes i and j in the BB-array and thus only a constant time cost is required to report the MBB. To sum up, we require $O(m^2 + C)$. □

Lemma 6 *The time complexity of approximate distance computation by using BB-array is less than that by using BB-tree.*

Proof *We analyze their time complexities in the following. The BB-tree needs* $O(m + C \cdot (\log m + 1))$ *and the BB-array needs* $O(m^2 + C)$.

$$\begin{aligned}&m + C \cdot (\log m + 1) - (m^2 + C) > \\ &C \cdot (\log m + 1) - (m^2 + C) = C \cdot \log m - m^2\end{aligned}$$

As $\log m \geq 1$ *and* $C \gg m$ *(the total number of candidates) for large datasets, we have* $C \cdot \log m > m^2$. *The cost of BB-tree is the average time complexity, while the cost of BB-array is the worst time complexity. Consequently, the BB-array outperforms the BB-tree.* □

7 Experimental Evaluation

We implement the proposal in C/C++ and perform the evaluation in SECONDO [8]. A desktop PC (Intel(R) Core(TM) i7-4770CPU, 3.4GHz, 4GB memory, 2TB hard

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3100650, IEEE Transactions on Knowledge and Data Engineering

disk) running Suse Linux 13.1 (32 bits, kernel version 3.11.6) is used. We use real GPS records of taxis from Beijing and Shanghai [1], named BTAXI and STAXI, respectively. We develop a tool to generate attributes for Beijing taxis. For STAXI, each GPS record is associated with an attribute indicating the company id. Table 3 summarizes the dataset statistics. For each attribute, the value is randomly selected from the domain. The CPU time and I/O accesses are used as performance metrics and the results are averaged over 20 runs.

TABLE 3
Datasets and parameter settings

Name	#GPS Records	$\|\mathcal{O}\|$	$\|A\|$	$dom(A)$	X range	Y range
BTAXI	235,634,511	4,220,435	10	[1, 151]	21-119,958	0-119,653
STAXI	202,919,952	6,280,600	1	[1, 4]	0-99,749	0-99,980
Query settings						
$\|Q_a\|$: {1, 2, 3, 4, 5}			d (km): {1, 5, 10, 20, 50}			

7.1 Setup

In the system, the fanout of a GR^2-tree node is 62. Such a value is determined by the page size. In the attribute structure R_{att}, each tuple defines a bitmap recording entries of a node containing an attribute value. We set the length of a bit array to be 32, that is, using a 32-bit integer. As a result, each bit maps $\lceil \frac{62}{32} \rceil = 2$ entries.

The grid granularity δ. We evaluate the performance affected by δ and report the query cost in Figure 11. According to Lemma 1, the optimal granularities for BTAXI and STAXI are $\delta^* = 11$ and $\delta^* = 15$, respectively. The number of partitions over time are $K = 259$ and $K = 446$, respectively. The experimental results confirm that our setting achieves the best performance.

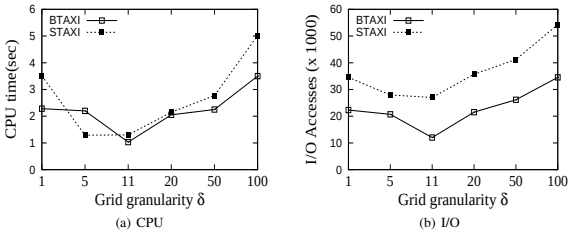


Fig. 11. Effect of δ^* for BTAXI and STAXI

The effect of establishing leaf nodes. As expected, the procedure of accessing leaf nodes does not incur accessing non-leaf nodes and require less CPU time and I/O accesses, as reported in Figure 12.

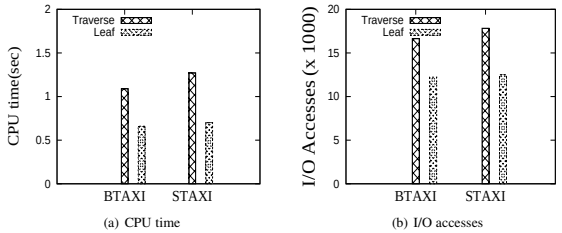


Fig. 12. Traverse the GR^2-tree versus Access leaf nodes

BB-array versus BB-tree. Part of BTAXI is chosen as the testing dataset (2,888,278 GPS records and 44,653 trajectories). The experimental results demonstrate up to an order of magnitude speed up by BB-array, as shown in Figure 13(a). Since the computation is executed many times in the query procedure (81,878), the overall running time is reduced by half. As both BB-tree and BB-array perform approximate distance computations, the relative error is calculated, that is the deviation between the exact distance and the approximate distance. The relative errors are 6% and 8% for BB-tree and BB-array, respectively. As a further step, we demonstrate the effect of the array size on the performance, as illustrated in Figure 13(b). The array size has little effect on the efficiency and thus in the following we set $B = \lceil m \cdot 0.38 \rceil$.

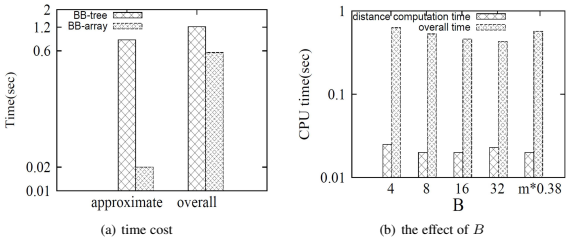


Fig. 13. BB-tree and BB-array

7.2 Performance evaluation

We perform the evaluation by comparing our method with six baseline methods in terms of scalability and efficiency: (1) **3-D R-tree**, (2) **RIB** [28], (3) **4-D R-tree**, (4) **IOC-Tree** [12], (5) **HAGI** [19] and (6) **Att-online,** an attribute index is built to at first collect trajectories containing Q_a. Then, a 3D R-tree is built on-the-fly to perform the evaluation.

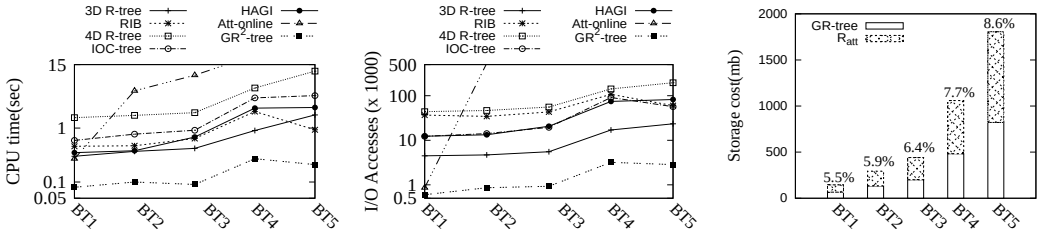
7.2.1 Scalability evaluation

Scaling the number of trajectories. Different subsets of BTAXI are selected, as summarized in Table 4. The performance result is reported in Figure 14. When the data size grows, the costs of all methods rise proportionally, but our method outperforms baseline methods by a factor of 5-50x on the largest dataset. The method Att-online is only competitive for a small dataset but the performance degrades significantly for large datasets. This is because the attribute predicate is not selective and thus a large number of trajectories are returned to build the index. We provide the storage cost of GR^2-tree and the ratio of the index size to the data size.

TABLE 4
Datasets for scaling $|\mathcal{O}|$

Name	$\|\mathcal{O}\|$	$\|A\|$	$dom(A)$	Size($\mathcal{O}$) (mb)
BT1	533,635	10	[1, 151]	2,631
BT2	1,009,579			4,983
BT3	1,424,273			7,041
BT4	2,757,312			13,678
BT5	4,220,435			20,910

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3100650, IEEE Transactions on Knowledge and Data Engineering



(a) CPU time (b) I/O accesses (c) The storage cost

Fig. 14. Scaling $|\mathcal{O}|$

Scaling data attributes. To investigate the effect of attributes on the performance, we choose the largest number of trajectories and scale (i) the number of attributes $|A|$ and (ii) the domain *dom*(A), as reported in Table 5.

TABLE 5
The settings of $|A|$ and dom(A)

$\|A\|$	2	5	10	15	20
dom(A)	[1, 43]	[1, 74]	[1, 211]	[1, 322]	[1, 861]
$\|A\|$ = 1					
[1, 5], [1, 20], [1, 50], [1, 100], [1, 200], [1, 500]					

The results are reported in Figures 15 and 16. The performance decreases when $|A|$ increases (also *dom*(A)) but enhances if $|A|$ is set by 1 and *dom*(A) is enlarged. This is because the attribute predicate becomes quite selective for a single attribute with a large domain. In this setting, RIB is slightly better than our solution, also when a small number of attribute values are defined. The index is built on trajectories grouped by attribute values and a good locality is achieved in terms of attributes. However, the performance suffers from dealing with multiple attributes as it is difficult to achieve a good data locality. Our method is superior to other methods in most settings for multiple attributes. The 3-D R-tree does not prune the search space on attribute, and thus a large number of candidates may be returned, deteriorating the efficiency. The 4-D R-tree enlarges the data set $|A|$ times. This increases the index overhead and also complicates the evaluation. Furthermore, the method does not achieve good locality for multiple attributes.

The IOC-tree associates attributes with all locations. Consequently, each octree maintains all points of a trajectory. This significantly increases the index storage overhead, deteriorating the query performance. HAGI defines a *loose* bound by *min* and *max* values and may work well for one attribute. However, the method will process all trajectories whose values are within the bound but are not equal to the query. The bound does not make sense if several attributes are defined because *min* and *max* values may be from different attributes. The scope of the method Att-online is limited as the performance is only competitive in a few settings, e.g., $|A|$ = 1 and *dom*(A) = 500. We will not include the method in the following evaluation. We also report the storages of GR-tree and R_{att}, respectively, see Figures 15(c) and 16(c). One can see that varying the attribute setting will only incur the variation of R_{att}'s storage.

7.2.2 Efficiency study

Varying $|Q_a|$. We perform the evaluation by varying the number of query attributes. The results, as reported in Figure 17, demonstrate that our method substantially outperforms baseline methods in all settings. When $|Q_a|$ increases, the performance becomes better as the attribute predicate is more selective.

Varying the distance d. We report the performance evaluation affected by d in Figures 19 and 20. When d increases, the performance degrades as expected due to more data being processed. The advantage of our method is significant in BTAXI, while some baseline methods are competitive in STAXI. That is, RIB achieves a good selectivity when there is only one attribute. The result is consistent with scaling the number of attributes.

Effect of $T(o_q)$. We do the evaluation by choosing query trajectories with different distributions of time periods: *short time*, *random* and *long time*. For example, we select 20 trajectories with the minimum time period as *short time* query trajectories. The results are reported in Figure 18. One can see that the costs grow proportionally when $T(o_q)$ increases, but our method achieves the best performance in all settings. Furthermore, the performance deviates significantly for long trips.

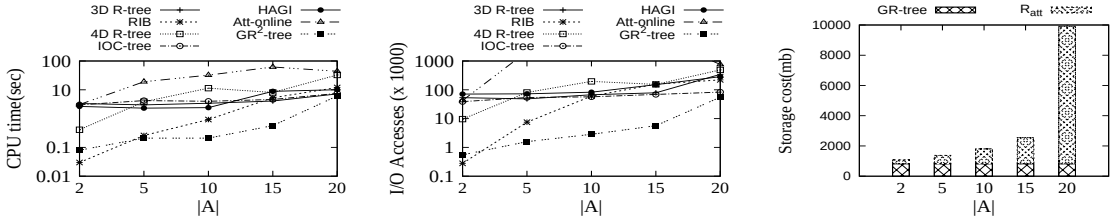
Memory costs. We report the average memory cost of auxiliary structures $\mathcal{C}^3$, BB-array and GR-tree in Table 6. When d increases, more cells will be involved during the evaluation and the storage of $\mathcal{C}^3$ rises accordingly. The storage of the BB-array depends on the query trajectory and thus is not sensitive to d. We measure the memory cost of GR-tree in terms of the number of accessed nodes during the query evaluation. The largest query distance d = 50 km incurs around 12mb memory cost.

TABLE 6
The memory costs of $\mathcal{C}^3$, BB-array and GR-tree

d (km)	1	5	10	20	50
$\mathcal{C}^3$ (kb)	0.27	0.28	0.8	4.56	28.52
BB-array (kb)	18.17	10.45	15.62	38.9	24.45
B	15	15	17	23	20
GR-tree(×4kb)	72	299	1,292	1614	2,760

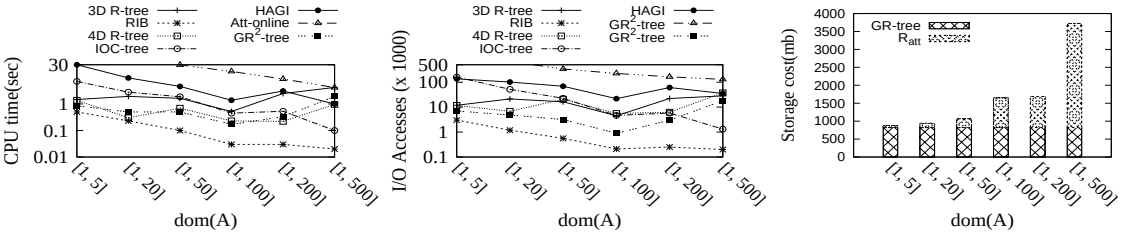
7.2.3 Updating evaluation

The performance is evaluated by scaling the number of new trajectories. We build the historical database on part of the dataset and take the rest as new trajectories, denoted by $\mathcal{O}_u$. As reported in Figure 21(a), $|\mathcal{O}|$ increases in



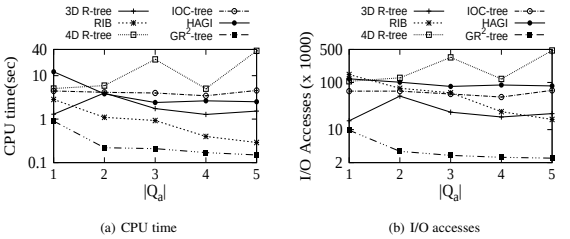
(a) CPU time (b) I/O accesses (c) The storage cost

Fig. 15. Scaling $|A|$ and *dom*(A)



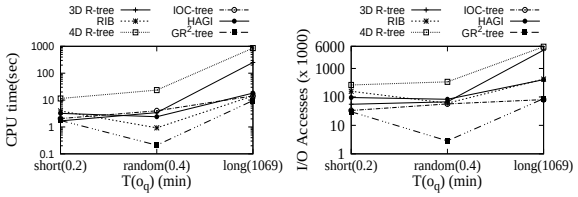
(a) CPU time (b) I/O accesses (c) The storage cost

Fig. 16. Scaling dom(A) ($|A|$ = 1)



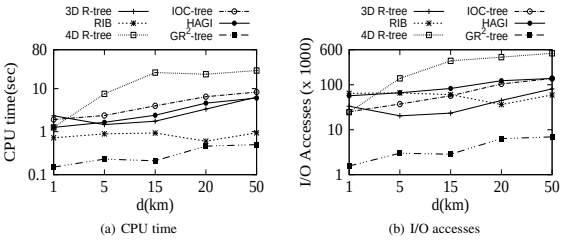
(a) CPU time (b) I/O accesses

Fig. 17. The effect of $|Q_a|$ (BTAXI, d = 10km)



(a) CPU time (b) I/O accesses

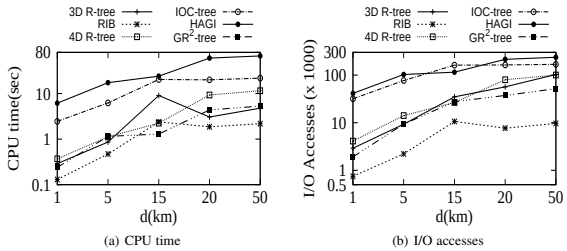
Fig. 18. The effect of query time (BTAXI, d = 10km, $|Q_a|$ = 3)



(a) CPU time (b) I/O accesses

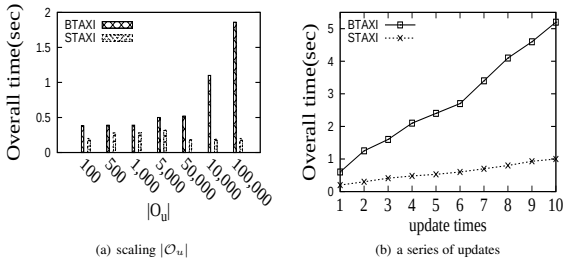
Fig. 19. The effect of d (BTAXI, $|Q_a|$ = 3)

several orders of magnitude, but the updating cost only rises marginally. The cost of BTAXI is higher than that of STAXI due to the number of attributes. We also perform a series of updates, each of which processes 50,000 trajectories. The overall update time is measured and the result is reported in an accumulated way, as shown in Figure 21(b). The time cost increases slightly.



(a) CPU time (b) I/O accesses

Fig. 20. The effect of d (STAXI, $|Q_a|$ = 3)



(a) scaling $|\mathcal{O}_u|$ (b) a series of updates

Fig. 21. The performance of updating GR^2-tree

8 Conclusions

We studied multi-attribute trajectories to enrich the data representation of trajectory data. A new query is proposed to simultaneously evaluate the data in terms of spatio-temporal and attribute predicates. We develop an optimal data partition method and build index structures with efficient query algorithms. An approximate distance calculation method is proposed accompanied with thorough theoretical analysis. Extensive experimental results demonstrated that our method significantly outperforms alternative methods. The future work is to consider join queries on multi-attribute trajectories.

Acknowledgment. This work is supported by NSFC under grants 61972198, Natural Science Foundation of Jiangsu Province of China under grants BK20191273. Zhifeng Bao is supported in part by ARC DP200102611.

REFERENCES

[1] http://factory.datatang.com/en/ (2019).

[2] J. Bercken, B.Seeger, and P.Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB*, pages 406–415, 1997.

[3] R. Cai, Z. Lu, L. Wang, Z. Zhang, T. Z. J. Fu, and M. Winslett. DITIR: distributed index for high throughput trajectory insertion and real-time temporal range query. *Proc. VLDB Endow.*, 10(12):1865–1868, 2017.

[4] D.W. Choi, J. Pei, and T. Heinis. Efficient mining of regional movement patterns in semantic trajectories. *PVLDB*, 10(13):2073–2084, 2017.

[5] J. Dai, B. Yang, C. Guo, C. S. Jensen, and J. Hu. Path cost distribution estimation using trajectory data. *PVLDB*, 10(3):85–96, 2016.

[6] Y. Fang, R. Cheng, W. Tang, S. Maniu, and X. S. Yang. Scalable algorithms for nearest-neighbor joins on big trajectory data. In *ICDE*, pages 1528–1529, 2016.

[7] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Algorithms for nearest neighbor search on moving object trajectories. *GeoInformatica*, 11(2):159–193, 2007.

[8] R. H. Güting, T. Behr, and C. Düntgen. SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.*, 33(2):56–63, 2010.

[9] R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *VLDB Journal*, 19(5):687–714, 2010.

[10] R. H. Güting, F. Valdés, and M.L. Damiani. Symbolic Trajectories. *ACM Transactions on Spatial Algorithms and Systems*, 1(2):Article 7, 2015.

[11] R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, and et al. A foundation for representing and querying moving objects. *ACM TODS*, 25(1):1–42, 2000.

[12] Y. Han, L. Wang, Y. Zhang, W. Zhang, and X. Lin. Spatial keyword range search on trajectories. In *DASFAA*, pages 223–240, 2015.

[13] Q. Hao, L. Chen, F. Xu, and Y. Li. Understanding the urban pandemic spreading of COVID-19 with real world mobility data. In *KDD*, pages 3485–3492, 2020.

[14] Y. Luo, W. Li, T. Zhao, X. Yu, L. Zhang, G. Li, and N. Tang. Deeptrack: Monitoring and exploring spatio-temporal data - A case of tracking COVID-19 -. *Proc. VLDB Endow.*, 13(12):2841–2844, 2020.

[15] C. Parent, S. Spaccapietra, C. Renso, and et al. Semantic trajectories modeling and analysis. *ACM Comput. Surv.*, 45(4):42, 2013.

[16] Z. Shang, G. Li, and Z. Bao. DITA: distributed in-memory trajectory analytics. In *SIGMOD*, pages 725–740, 2018.

[17] H. Su, K. Zheng, H. Wang, J. Huang, and X. Zhou. Calibrating trajectory data for similarity-based analysis. In *SIGMOD*, pages 833–844, 2013.

[18] H. Su, K. Zheng, K. Zeng, J. Huang, S. W. Sadiq, N. J. Yuan, and X. Zhou. Making sense of trajectory data: A partition-and-summarization approach. In *ICDE*, pages 963–974, 2015.

[19] Y. Su, Y. Wu, and A. L. P. Chen. Monitoring heterogeneous nearest neighbors for moving objects considering location-independent attributes. In *DASFAA*, pages 300–312, 2007.

[20] Y. Tong, Y. Chen, Z. Zhou, L. Chen, J. Wang, Q. Yang, J. Ye, and W. Lv. The simpler the better: A unified approach to predicting original taxi demands based on large-scale online platforms. In *ACM SIGKDD*, pages 1653–1662, 2017.

[21] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu. A unified approach to route planning for shared mobility. *PVLDB*, 11(11):1633–1646, 2018.

[22] G. Trajcevski and P. Scheuermann. Triggers and continuous queries in moving objects database. In *DEXA*, pages 905–910, 2003.

[23] F. Valdés and R. H. Güting. A framework for efficient multi-attribute movement data analysis. *VLDB J.*, 28(4):427–449, 2019.

[24] F. Valdés and R. Hartmut Güting. Index-supported pattern matching on tuples of time-dependent values. *GeoInformatica*, 21(3):429–458, 2017.

[25] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, M. Sanderson, and X. Qin. Answering top-k exemplar trajectory queries. In *ICDE*, pages 597–608, 2017.

[26] S. Wang, Z. Bao, J. Shane Culpepper, T. Sellis, and G. Cong. Reverse k nearest neighbor search over trajectories. *IEEE Trans. Knowl. Data Eng.*, 30(4):757–771, 2018.

[27] S. Wang, Z. Bao, J. Shane Culpepper, Z. Xie, Q. Liu, and X. Qin. Torch: A search engine for trajectory data. In *SIGIR*, pages 535–544, 2018.

[28] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE Trans. Knowl. Data Eng.*, 24(10):1889–1903, 2012.

[29] D. Xie, F. Li, and J. M. Phillips. Distributed trajectory similarity search. *PVLDB*, 10(11):1478–1489, 2017.

[30] Z. Yan, D. Chakraborty, C. Parent, S. Spaccapietra, and K. Aberer. Semitri: a framework for semantic annotation of heterogeneous trajectories. In *EDBT*, pages 259–270, 2011.

[31] Z. Yan, D. Chakraborty, C. Parent, S. Spaccapietra, and K. Aberer. Semantic trajectories: Mobility data computation and annotation. *ACM TIST*, 4(3):49:1–49:38, 2013.

[32] C. Zhang, J. Han, L. Shou, J. Lu, and T. F. La Porta. Splitter: Mining Fine-Grained Sequential Patterns in Semantic Trajectories. *PVLDB*, 7(9):769–780, 2014.

[33] J. Zhang, B. Tang, and M. L. Yiu. Fast trajectory range query with discrete frechet distance. In *EDBT*, pages 634–637, 2019.

[34] B. Zheng, N. J. Yuan, K. Zheng, X. Xie, S. W. Sadiq, and X. Zhou. Approximate keyword search in semantic trajectory database. In *ICDE*, pages 975–986, 2015.

[35] K. Zheng, S. Shang, N. J. Yuan, and Y. Yang. Towards efficient search for activity trajectories. In *ICDE*, pages 230–241, 2013.

[36] K. Zheng and H. Su. Go beyond raw trajectory data: Quality and semantics. *IEEE Data Eng. Bull.*, 38(2):27–34, 2015.

Jianqiu Xu is a professor in Nanjing University of Aeronautics and Astronautics, China. His research interests include spatial databases and moving objects databases. He has served on the program committees for conferences such as KDD, DASFAA and MDM.

Zhifeng Bao is an associate professor with the RMIT University in Australia. His research interests include data usability, spatial database, data integration, and cleaning. He has served on the program committees for conferences such as SIGMOD, PVLDB, ICDE and KDD.

Hua Lu is a professor with the Department of Department of People and Technology, Roskilde University, Denmark. His research interests include database and data management, and geographic information systems. He has served on the program committees for conferences such as PVLDB, ICDE and KDD.