Geoinformatica (2018) 22:723–766
https://doi.org/10.1007/s10707-018-0326-5

Continuous *k* nearest neighbor queries over large multi-attribute trajectories: a systematic approach

Jianqiu Xu[1] · **Ralf Hartmut Güting**[2] · **Yunjun Gao**[3]

Received: 11 October 2017 / Revised: 8 April 2018 / Accepted: 28 June 2018 /
Published online: 11 July 2018

Abstract We study *multi-attribute trajectories* by combining standard trajectories (i.e., a sequence of timestamped locations) and descriptive attributes. A new form of continuous *k* nearest neighbor queries is proposed by integrating attributes into the evaluation. To enhance the query performance, a hybrid and flexible index is developed to manage both spatio-temporal data and attribute values. The index includes a 3D R-tree and a composite structure which can be popularized to work together with any R-tree based index and Grid-based index. We establish an efficient mechanism to update the index and define a cost model to estimate the I/Os. Query algorithms are proposed, in particular, an efficient method to determine the subtrees containing query attributes. Using synthetic and real datasets, we carry out comprehensive experiments in a prototype database system to evaluate the efficiency, scalability and generality. Our approach gains more than an order of magnitude speedup compared to three alternative approaches by using 1.8 millions of trajectories and hundreds of attribute values. The update performance is evaluated and the cost model is validated.

✉ Jianqiu Xu
jianqiu@nuaa.edu.cn

Ralf Hartmut Güting
rhg@fernuni-hagen.de

Yunjun Gao
gaoyj@zju.edu.cn

[1] Nanjing University of Aeronautics and Astronautics, Nanjing, China

[2] FernUniversität in Hagen, Hagen, Germany

[3] ZheJiang University, Hangzhou, China

Keywords Trajectories · Multi-attribute · Continuous queries · Nearest neighbors · Index structure · Update

1 Introduction

1.1 Motivation

Trajectory data, keeping track of historical movements of moving objects such as vehicles and ships, is becoming ubiquitous due to the widespread use of GPS devices. Such data that records geographical locations changing over time is of crucial importance for emerging applications, e.g., route recommendation [9], tracking [28], monitoring [60], crowdsourcing [47, 49], to name but a few.

Despite tremendous efforts made on studying trajectory databases, proposals in the literature mainly deal with standard trajectories [33, 50, 66], i.e., a sequence of timestamped geo-locations, and the majority of queries are limited to the spatio-temporal evaluation such as range queries [51], nearest neighbors [21] and convoys [27]. In the real world, typical moving objects such as vehicles and persons are associated with pieces of descriptive information. The database system should fully represent moving objects and allow users to query objects with extensive knowledge to better understand the movement and users' behavior.

As a fundamental step towards that, we investigate a new form of trajectories called *multi-attribute trajectories*, each of which consists of a standard trajectory and a set of attribute values. Standard trajectories have been well established in the literature. Attributes have various semantics according to applications. The combination allows users to query trajectories by specifying certain attributes. We study continuous $\underline{k}$ nearest neighbor queries over multi-attribute trajectories, CkNN_MAT for short. Given a standard trajectory, the number of neighbors k and attribute values, CkNN_MAT returns k trajectories at each defined time instant, each of which (i) contains query attribute values and (ii) belongs to k nearest neighbors of the query trajectory. Returned objects should fulfill the condition: (i) *attribute consistency* and (ii) *time-dependent distance closeness*. Distances between moving objects vary over time, and thus returned objects change at certain time points, complicating the evaluation. The Euclidean distance is measured.

Consider a running example. A database stores five vehicle trajectories $\{mo_1, mo_2, mo_3, mo_4, mo_5\}$ in which each contains a sequence of timestamped locations and two attribute values from domains *COLOR* = {RED, SILVER, BLACK} and *BRAND* = {BENZ, VW, TOYOTA}, respectively, as illustrated in Fig. 1. An interesting query "*Continuously report the nearest SILVER VW to* mo_4" is issued. CkNN_MAT returns $\{([t_1, t_2], mo_5), ([t_2, t_4], mo_3)\}$ indicating the key aspect that only objects fulfilling the attribute condition will be evaluated on the time-dependent closeness. Although mo_1 and mo_2 are closer than mo_3 and mo_5 to the query trajectory, they do not contain (SILVER VW) and will be excluded. The query cannot be answered in standard trajectory databases because the attribute is not defined and the system does not know which trajectories are SILVER VWs. If attributes are ignored, the query becomes traditional nearest neighbor queries [16, 21, 46] and will report $\{([t_1, t_2], mo_2), ([t_2, t_3], mo_1)\}$.

The combined data enriches the representation and exposes semantics that is orthogonal to location data, which most semantic-enriched trajectories focus on [2, 61, 62]. Multi-attribute trajectories raise a range of novel queries by integrating the attribute value into the evaluation. We do not deal with attributes such as speed and acceleration that can be calculated from the motion function and are not necessarily to be defined.

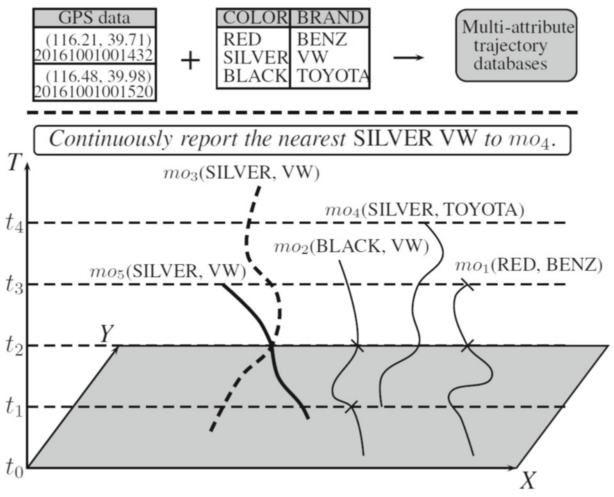


Fig. 1 The motivating example

1.2 Previous works and challenges

In the current state-of-the-art, standard trajectories with additional information have been investigated, e.g., semantic trajectories [36, 61], activities trajectories [64], and transportation modes [56]. A semantic/activity trajectory is typically defined to be a sequence of locations attached with semantic labels, e.g., *park*, *restaurant*. In principle, semantic/activity trajectories is an enriched version of standard trajectories in terms of semantic labels describing locations. This is orthogonal to multi-attribute trajectories. We point out major differences in this section and provide a thorough comparison in related work.

(i) Attributes represent a range of aspects of the objects and provide a full picture of moving objects, as opposed to semantics limited to locations. This will support a different (even broader) range of applications.
(ii) Semantic locations are sparsely defined because among a person's trajectory a few locations have semantics. Attributes are location-independent and associated with the complete trajectory. They are not derived from timestamped locations or the geographical environment.
(iii) Semantic trajectories cope with similarity search or ranking queries rather than continuous queries with the exact match on attribute values, leading to different tasks when developing the index. Semantic trajectories are grouped in terms of locations and semantics, but attributes are not related to locations.

Efficiently answering CkNN_MAT is not trivial. Standard trajectory indexes such as R-tree [24], TB-tree [37], SETI [6] and TrajStore [35] only deal with the spatio-temporal data without managing attributes. We cannot use the index to perform the selection on attribute values. Furthermore, the pruning technique of *min* and *max* distances cannot be applied if trajectories attributes are not determined. The trajectory subset containing query attributes dynamically changes according to the query and cannot be pre-computed. False dismissals will occur if we perform the pruning without the awareness of attribute values. Consider the example in Fig. 1. Although mo_2 has smaller distances to mo_4 than mo_3, we cannot prune mo_3 because mo_2 does not contain (SILVER, VW).

CkNN_MAT could be answered by employing an attribute index. We evaluate the attribute condition to receive trajectories containing query attributes and then proceed to processing standard trajectories. However, this method is limited in scope and inherently suffers from the performance issue. Standard trajectories will be processed by either performing a sequential scan or accessing an on-line built index. If the attribute predicate is selective, the query cost may be acceptable because a small dataset is processed. If the attribute predicate has a bad selectivity, a large number of trajectories will be returned. Both the sequential scan and building an on-line index incur high CPU and I/O costs. Furthermore, creating an index for each query at runtime causes extra storage space.

To sum up, the main challenge is how to let the index effectively and efficiently preserve the spatio-temporal proximity, and also maintain attributes. Individually managing each part can be easily achieved but does not achieve an optimal performance. We aim to develop a general structure that can be used for a range of queries rather than ad-hoc methods for a particular query workload.

1.3 The solution

We start by modeling attributes and then use a relational interface to integrate standard trajectories and attributes. This ensures that existing operators on standard trajectories can be leveraged, avoiding starting from scratch. To reduce approximations of standard trajectories, we identify small pieces of movements and pack them to have a compact dataset to build the index. To manage multi-attribute trajectories, we propose an index structure made up of a 3D R-tree and a composite structure BAR. The 3D R-tree is to preserve the spatio-temporal proximity, and BAR built on top of R-tree is to manage attribute values.

We do not tightly integrate attribute values into the spatio-temporal index, but offer a versatile approach such that BAR is compatible with most spatio-temporal indexes and can be discarded if attributes are not defined. We provide a dynamic structure to support tracking incoming data and also define a cost model to estimate the update costs.

We design a query framework following the *filter-and-refine* strategy. The filter accesses BAR to determine the R-tree nodes fulfilling the attribute condition in which an efficient access method is proposed to determine the nodes and a thorough analysis is provided. We traverse the R-tree from top to bottom level during which one evaluates both attribute and spatio-temporal predicates for each accessed node. The filter returns a set of candidate trajectories, each of which contains query attributes and approximately belongs to k nearest neighbors. The refinement runs the accurate distance computation. The proposal is implemented in an extensible database system SECONDO [20] to have a practical result and achieve the system development. The contributions of the paper are summarized as follows:

- We offer insight into multi-attribute trajectories by providing the data representation and proposing a new query CkNN_MAT. A query expression is defined to formulate attribute values.
- We propose a hybrid and flexible index on a compact dataset by packing standard trajectories and provide an optimal access method. The storage cost of the index is analytically provided.
- Employing a light-weight structure, we develop a method to efficiently update the index to keep track of incoming multi-attribute trajectories and establish a cost model to estimate update I/Os.
- Efficient query algorithms are developed to answer CkNN_MAT in which different forms of query attributes are supported.

– Using large real and synthetic datasets, an extensive experimental study is conducted to demonstrate the significant performance advantage of our method over three baseline methods in terms of efficiency and scalability.

The rest of the paper is organized as follows: The problem is defined in Section 2. We present the index structure in Section 3 and turn to the update issue in Section 4. Query algorithms are introduced in Section 5. Section 6 reports the evaluation, Section 7 discuses the generality of the proposal, and Section 8 surveys the related work, followed by conclusions in Section 9.

2 Problem definition

2.1 Standard trajectories

A standard trajectory is typically modeled by a function from time to 2D space. In a database system, a discrete model is implemented, that is to represent the trajectory by a sequence of so-called *temporal units*, as illustrated in Fig. 2. Each unit records start and end locations during a time interval, and locations between start and end locations are estimated by interpolation. A data type called $\underline{mpoint}$ is defined. The comprehensive framework refers to [15, 22].

Definition 1 Standard trajectories
$D_{\underline{mpoint}} = \{< u_1, ..., u_n > \mid \forall i \in [1, n], u_i = (t_1, t_2, p_1, p_2), t_1, t_2 \in \underline{instant}, p_1, p_2 \in \mathbb{R}^2 \}$

2.2 An integration

Let $A = \{A_1, ..., A_d\}$ denote d attributes to be combined with standard trajectories, $d \geq 1$. We refer to the domain of each attribute as $dom(A_i)$ ($i \in \{1, ..., d\}$) and define $\forall\, i \in \{1,...,d\}$: $dom(A_i) \subset \mathbb{Z}^+$. A d-attribute object is represented by the following data type.

Definition 2 Multi-attributes
$D_{\underline{att}} = \{(a_1, ..., a_d) \mid a_i \in \text{dom}(A_i), i \in \{1, ..., d\}\}$

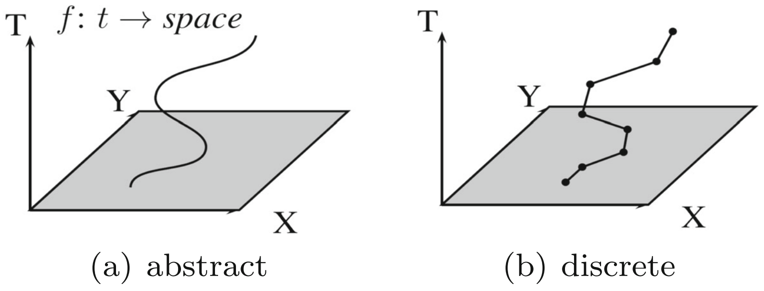


Fig. 2 Standard trajectory representation

The overall domain is denoted by $dom(\text{A}) = \bigcup_{i=1}^{d} dom(A_i)$. Attribute semantics is set according to real applications. In the running example, we have two attributes $A_1 = COLOR$ and $A_2 = BRAND$ with domains $dom(A_1)$ = {RED, SILVER, BLACK} and $dom(A_2)$ = {BENZ, VW, TOYOTA}, respectively. We use a composite data model $\mathcal{D}$(Trip; Att) to represent a multi-attribute trajectory database, where *Trip* denotes standard trajectories and *Att* denotes multi-attributes. The data model is translated to a relation with the schema (*Id*: $\underline{int}$, *Trip*: $\underline{mpoint}$, *Att*: $\underline{att}$) by embedding data types $\underline{mpoint}$ and $\underline{att}$ as relational attributes, as shown in Table 1. The advantage of the relational interface is that (i) it allows combining heterogeneous data models; and (ii) existing operators on standard trajectories can be leveraged, benefiting the system development.

2.3 Queries

We incorporate attribute values into the evaluation and define the query expression

Definition 3 Attribute query expression
$Q_a = (a_1, ..., a_d), a_i \in dom(A_i) \text{ or } a_i = \perp.$

The attribute expression for "*Continuously report the nearest SILVER VW to* mo_4" is formed by Q_a = (SILVER, VW). We use the notation $\perp$ for an undefined value. If the user does not define any attribute, the query regresses to traditional nearest neighbor query. Given a d-attribute object $o \in D_{\underline{att}}$, let $o.A_i$ refer to the value of the attribute A_i. An operator called **contain** is defined.

contain(o, Q_a) returns *true* if $\forall\, a_i \in Q_a$: $o.A_i = a_i$ or $a_i = \perp$.

Let $dist(t, Trip_1, Trip_2)$ return the distance between two standard trajectories at a time point t. CkNN_MAT is formally defined as follows.

Definition 4 CkNN_MAT
Given a query standard trajectory mq, an integer k, and Q_a, CkNN_MAT returns k trajectories denoted by $\mathcal{D}' \subseteq \mathcal{D}$ at each time $t \in T(mq)$ such that $\forall\, mo' \in \mathcal{D}'$: **contain**($mo'.Att$, $Q_a) \wedge \forall mo \in \mathcal{D} \setminus \mathcal{D}' : dist(t, mo', mq) < dist(t, mo, mq)$

Referring to Fig. 1, CkNN_MAT returns mo_5 during $[t_1, t_2]$ and mo_3 during $[t_2, t_3]$. In the following, we will use the notation t for a time point and also a time interval, and explicitly indicate the meaning. Users can also define multiple values of an attribute, e.g., "*Continuously report the nearest SILVER or BLACK VW to* mo_4". We generalize the expression of the query attribute.

Definition 5 An extension
$Q_a = (X_1, ..., X_d)$, $X_i \subseteq dom(A_i)$ or $X_i = \emptyset$.

Table 1 Integrate standard trajectories and attributes

Id	*Trip*	*Att*
mo_1	location + time	(RED, BENZ)
mo_2	location + time	(BLACK, VW)
mo_3	location + time	(SILVER, VW)
mo_4	location + time	(SILVER, TOYOTA)
mo_5	location + time	(SILVER, VW)

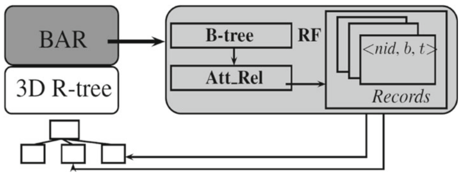


Fig. 3 Index architecture

At the concept level, $Q_a = (X_1, ..., X_d)$ defines the component for each attribute over $\{A_1, ..., A_d\}$, in which X_i is a set of attribute values. The multi-value query "*SILVER or BLACK VW*" is formed by Q_a = ({SILVER, BLACK}, {VW}). At the implementation level, the query is defined by a relation in which a tuple supports multi-valued attributes.

The operator **contain** is extended accordingly:

contain(o, Q_a) returns *true* if $\forall\ X_i \in Q_a$: $o.A_i \in X_i$ or $X_i = \emptyset$.

Table 2 summarizes notations frequently used in the following paper.

3 The index

3.1 An overview

The proposed index structure consists of a 3D R-tree and BAR, as shown in Fig. 3. The 3D R-tree that serves as indexing standard trajectories is a height balanced tree. Each node contains an array of entries, each of which couples i) a pointer pointing to a subtree or an object with ii) a rectangle that bounds data objects in the subtree. BAR is a composite structure that includes a B-tree, a relation *Att_Rel* and a record file *RF*, and will be elaborated in Section 3.3.

We build BAR on top of the 3D R-tree by extracting attribute values from multi-attribute trajectories. The structure builds the connection between attribute values and R-tree nodes, and enables us to know attribute values in a sub-tree. In a leaf node, each entry stores a pointer pointing to a tuple in the relation. We access the tuple to obtain the value. For a non-leaf node, attribute values are collected by performing the union on values from child nodes. BAR maintains attribute values in an efficient way such that we are able to fast settle the R-tree nodes that (i) contain query attributes and (ii) fall into the query time.

Table 2 Summary of notations

Notation	Description
$\mathcal{D}$	Multi-attribute trajectory database
mo, mq	Multi-attribute trajectory, query standard trajectory
d	The number of attributes
$dom(A_i)$, $dom(A)$	The domain of A_i, the overall domain
Q_a	Query attribute expression
k	The number of nearest neighbors
t	A time point or interval

3.2 Packing standard trajectories

The R-tree is supposed to be built on sorted minimum bounding rectangles (MBRs) that approximate trajectories.

By observation we find that raw trajectories from GPS records contain a large number of *small* units due to short time intervals or slow movement.

In order to reduce the size of the dataset, we *pack* small pieces of movements to have less but larger units. Let u_i denote the unit extent in the ith dimension. The average extent over all units in the ith dimension is denoted by Δ_i. Then, the deviation of a unit is given as:

$$f(u) = \sum \frac{u_i}{\Delta_i}, i \in \{d_x, d_y, d_t\}. \quad (1)$$

A threshold *Bound* is defined to select *small* units. We remove duplicate values to overcome the impact of the number of small units and analytically estimate the lower bound.

$$\mathit{Bound} = \mathrm{Avg}(\mathrm{Unique}(\lfloor f(u) \rfloor)) \geq \mathrm{Avg}(\lfloor f(u) \rfloor) \approx 3.$$

Let U be the set of all temporal units and we define

$$u^* = \underset{u \in U}{\arg\max}\ \mathrm{Unique}(\lfloor f(u) \rfloor).$$

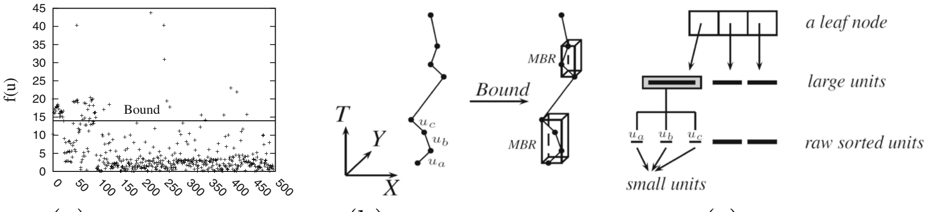
Not all values in {0, 1, ..., $\lfloor f(u^*) \rfloor$} may be defined and thus the upper bound is

$$\mathit{Bound} \approx \mathrm{Avg}(0 + ... + f(u^*)) \leq \frac{f(u^*)}{2}.$$

Using GPS records of ShangHai taxis [1], we calculate the unit deviation over 500 temporal units and report their values as well as *Bound* in Fig. 4. One can see that the majority of units have the derivation smaller than *Bound*. We pack successive small units of the same trajectory as one unit such that the deviation of the unit is larger than *Bound*. The overall number of trajectory approximations (MBRs) is greatly reduced, leading to a compact dataset to build the index.

Lemma 1 *Given $|U|$ temporal units, the number of MBRs is less than $\frac{3}{4} \cdot |U|$ after packing.*

Proof Let $U_s = \{u | u \in U \wedge f(u) < \mathit{Bound}\} \subset U$ be the set of small units. Suppose that the lower bound $\mathrm{Avg}(\lfloor f(u) \rfloor)$ is taken, leading to $|U_s| \approx \frac{|U|}{2}$. We have $\mathit{Bound} = \mathrm{Avg}(\mathrm{Unique}(\lfloor f(u) \rfloor)) \geq \mathrm{Avg}(\lfloor f(u) \rfloor)$ and therefore have $|U_s| \geq \frac{|U|}{2}$. Each $u \in U_s$ will be



(a) Sample data (b) Pack small units (c) Analysis

Fig. 4 Demonstrate packing

packed with at least one unit.[1] In the sequel, $|U_s|$ units will become $\frac{|U_s|}{2}$ units at most and the number of units after packing is $(|U|$ - $|U_s|) + \frac{|U_s|}{2} = |U|$ - $\frac{|U_s|}{2} \leq |U|$ - $\frac{|U|}{4} = \frac{3}{4} \cdot |U|$. $\square$

The packing can be treated as building the R-tree in a different way. We pack small pieces of trajectories to obtain large units and take them as the input for a leaf node. The index uses the same threshold as the bulk load method [4] to group units into one leaf node, guaranteeing the spatio-temporal locality. The defined *Bound* is the average value over Unique($f(u)$) and thus will not result in grouping units into a large extent. During the packing procedure, neither raw units are modified/simplified nor data is lost. We do not need extra storage space and maintain the same amount of original units. After packing, the R-tree is built by bulk load.

3.3 BAR

We need an appropriate structure to manage trajectory attributes in order to prune the data when traversing the index. One possible approach is to integrate attribute values into existing trajectory indexes. However, each multi-attribute trajectory is associated with a set of attribute values, leading to a complex task for a large number of trajectories. Modifying existing trajectory indexes will complicate the structures and also make them ad-hoc. We keep the original trajectory index and build a structure on top of that to manage attributes.

3.3.1 The Relation Att_Rel

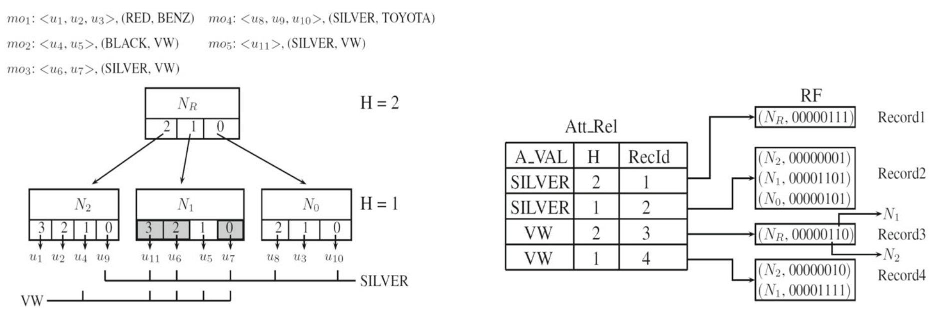
The key component in BAR is the relation Att_Rel that builds the connection between attribute values and R-tree nodes. The relation schema is defined as

Att_Rel(*A_VAL*: $\underline{int}$, *H*: $\underline{int}$, *RecId*: $\underline{int}$).

For each attribute value, we maintain a tuple for all nodes containing the value at the same height. The nodes are stored in a record. A tuple stores the attribute value, the height, and the record identifier. The relation is created as follows. First, for each $a \in dom(A)$ we traverse the R-tree in depth-first search to collect all nodes containing a and create an intermediate tuple for each node. We set a as the key and record the node height. Second, the intermediate tuples are grouped according to the height and a record is used to store all nodes containing a at each height. A tuple is created to store the record id. We repeat this procedure for all $a \in dom(A)$. The number of tuples in Att_Rel is $O(|dom(A)| \cdot H)$ in which $|dom(A)|$ is the overall number of attribute values and H is the R-tree height. We create a B-tree on Att_Rel by making a key combining A_VAL and H. Using example trajectories, we show the 3D R-Tree and BAR in Fig. 5.

Unique attribute values We need a unique key for each attribute value. The ideal case is that attribute domains do not overlap. In practice, it may be not possible to have non intersecting domains, but this problem can be solved. One can use a composite number to represent the attribute value. This is achieved by combining the attribute id and the value. In turn, a two-dimensional point (i, a) $(i \in [1, d], a \in dom(A_i))$ is formed. Then, a space-filling curve *Z-order* is used to map points of a two-dimensional space to one-dimensional

[1]Trajectories containing only a single unit are treated as dirty data and will be removed from the dataset. It is rare and impractical that two consecutive GPS records have a major deviation.



A_VAL	H	RecId
SILVER	2	1
SILVER	1	2
VW	2	3
VW	1	4

(a) 3D R-Tree on packed units (b) BAR (SILVER and VW)

Fig. 5 Exemplify the index structure

values. This is done by interleaving the binary coordinate values and is able to guarantee that attribute domains do not overlap. We prove this in Appendix.

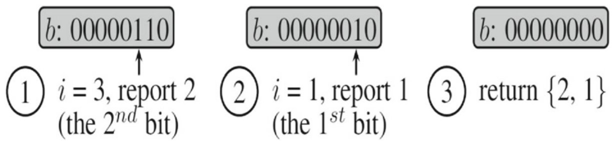
3.3.2 Record storage

Record format We maintain a list of items in each record. Each item is represented by a three-tuple: (nid, b, t), in which nid is the node id, b is a bitmap and t is a time interval. The bitmap represents the entries containing the attribute value in a node and t is the overall time of the entries. We make the design based on the observation that the number of entries containing an attribute value is no larger than the total number of entries, usually much smaller. Also, the number of entries containing an attribute value increases from leaf level to root level. This is because if a node contains the value, all its ancestor nodes will contain the value. Consider SILVER by referring to Fig. 5. Only one entry in N_2 contains SILVER but all entries in N_R contain SILVER. To efficiently settle the entries fulfilling the attribute condition, we access the bitmap instead of performing a linear scan over all entries.

Bitmap Let m denote the length of a collection of bit-vectors and E be the node *fan-out*. We perform the mapping between bitmaps and entries. There are two cases. (i) $m \geq E$, each bit maps to a unique entry. If the ith $\in [0, m)$ entry contains the value, we have $b[i] = 1$. Otherwise, $b[i] = 0$. (ii) $m < E$, each bit maps to a sequence of entries. The corresponding entries for the ith bit are calculated by $[i \cdot \lceil \frac{E}{m} \rceil, (i+1) \cdot \lceil \frac{E}{m} \rceil)$. We define $b[i] = 1$ if one of the entries contains the value. The bitmap index incurs little storage overhead and is efficient for processing data in small quantity due to the speed of bit-wise operations. The length m depends on the implementation (e.g., a 32-bit integer) and E is bounded by the *minimum* and *maximum* numbers of entries in a node. The bitmap is able to fast determine qualified entries for the intersection condition of several attributes. By defining $m = 8$, Fig. 5 gives the bitmap of each item in the record. Consider Q_a = (SILVER, VW). At $H = 2$, we access Records 1 and 3, and retrieve b = 00000111 for SILVER and b = 0000110 for VW, respectively. By performing the bitwise AND operation, the 0th entry is not defined and therefore N_0 will not be further considered. A data type is embedded into an relational to represent the records.

Definition 6 Record

$$D_{\underline{rec}} = \{((nid_1, b_1, t_1), ..., (nid_n, b_n, t_n)) | nid_i \in \underline{int}, b_i \in \underline{int}, t_i \in \underline{interval}\}$$



b: 00000110
b: 00000010
b: 00000000
1 i = 3, report 2 (the 2nd bit)
2 i = 1, report 1 (the 1st bit)
3 return {2, 1}

Fig. 6 Report defined bits

3.3.3 Querying the bitmap

In order to know the entries containing the query attribute we need to access the bitmap to report defined bits. A straightforward way is to apply a linear scan, leading to the time complexity $O(m)$. We propose an optimal method to boost the performance when the number of defined bits is smaller than m. Let $\mathcal{B} = < 2^0, 2^1, ..., 2^{m-1} >$ be a sequence of integers. Given a bitmap b, its defined bits are reported as follows: Step 1, by performing a binary search we find the smallest $2^i \in B$ such that $2^i \geq b$. Step 2, if $2^i = b$, we report i and terminate the search because the bit is already found and no more bit is to be reported. If $2^i > b$, we report i - 1 and update $b = b - 2^{i-1}$. Then, we repeat Steps 1-2 until b decreases to 0, during which bits are progressively reported from high to low positions. Figure 6 depicts the procedure of reporting b = 00000110 in Record 3.

We give the query algorithm in Algorithm 1. Let P denote the set of defined bit positions, initially empty. Two indexes s and e are used to define the sth and eth integers in $\mathcal{B}$. To find the smallest i such that $2^i \geq b$, the procedure performs a binary search and terminates when either b is equal to an integer in $\mathcal{B}$ (line 8) or $e = s + 1$ (line 14). In the former case, all bits are found already. In the latter case, the position of the highest bit is found and put into P (lines 16-17).

Algorithm 1 *ReportBits*(b)

```
1:  $\mathcal{B} \leftarrow < 2^0, 2^1, ..., 2^{m-1} >, P \leftarrow \oslash$;
2:  $s \leftarrow 0, e \leftarrow m$;
3:  if $|b| = m$ then $P \leftarrow \{0, 1, ..., m - 1\}$;
4:  else
5:      while $b > 0$ do
6:          while $s < e$ do
7:              $p \leftarrow (s + e) / 2$;
8:              if $\mathcal{B}[p] = b$ then
9:                  $P \leftarrow p, b \leftarrow b - \mathcal{B}[p]$;
10:                 break;
11:             else
12:                 if $\mathcal{B}[p] < b$ then $s \leftarrow p$;
13:                 else $e \leftarrow p$;
14:             if $e = s + 1$ then break;
15:         if $b > 0$ then
16:             $P \leftarrow P \cup \{s\}, b \leftarrow b - \mathcal{B}[s]$;
17:             $e \leftarrow s, s \leftarrow 0$;
18: return $P$;
```

To continue searching the bits, we update b as well as s and e. The baseline method to update s and e is to set $s \leftarrow 0$ and $e \leftarrow m$ after each iteration. Consequently, the binary search is repeatedly performed on the overall bit set. Motivated by the fact that b decreases after each iteration, the position of the next defined bit cannot be larger than s. Thus, we perform the update by setting $e \leftarrow s$ and $s \leftarrow 0$. The correctness is proved in the following.

Lemma 2 *If $2^s \leq b < 2^e$ and $e = s + 1$, the highest bit in b is s.*

Proof Let $s' \neq s$ be the highest bit and there are two cases:

(i) $s' > s$, that is $s' \in [e, m)$. Then, we have $b \geq 2^{s'} \geq 2^e$. This contradicts $b < 2^e$.
(ii) $s' < s$, that is $s' \in [0, s)$. Then, we have $2^s > b$ because s' is the highest bit. This contradicts $2^s \leq b$. □

Time complexity We need $O(\log m)$ to report the highest bit and denote the position by $p \in [0, m)$. To find the second highest bit, a binary search is performed, leading to $O(\log p)$. The iteration time depends on p. The smaller p is, the less iterations are needed. If $p \in [m/2, m)$, we need $\log p = \log m$ iterations. If $p \in [0, m/2)$, we achieve $\log m$ - 1 iterations. To report the ith highest bit, the iteration time is between [$\log m$ - (i - 1), $\log m$], depending on where the bit is located. To sum up, we have

Theorem 1 *Upper Bound*

$$O(|b| \log m)$$

Theorem 2 *Lower Bound*

$$O\left(\sum_{i=1}^{|b|}(\log m - (i-1)\right) = O(|b| \log m - (|b|-1)|b|/2)$$

Proof We perform a binary search to look for $|b|$ defined bits in $O(m)$. In the worst case, they are the $|b|$ highest bits and we need $O(\log m)$ for each iteration, leading to $O(|b| \log m)$. An optimal case is that we need $O(\log m$ - 1) for the second iteration when the position of the second highest bit is smaller than $\frac{m}{2}$. If the position of the second highest bit is $\leq \frac{m}{4}$, we need $O(\log m$ - 2) for the third iteration and so on. □

The binary search guarantees an optimal performance as long as $|b| \leq \lceil \frac{m}{\log m} \rceil$. The advantage may be not significant when the number of bits is large and the bits are mostly located at high positions. For example, if $|b| = m$, the upper bound is actually larger than the linear search performance, $O(|b| \log m) = O(m \log m) > O(m)$. If this happens, we just perform a linear search.

3.3.4 The storage

BAR includes the relation Att_Rel, the B-tree and the record file. The storage cost of Att_Rel is $O(|dom(A)| \cdot H)$, also the B-tree. The size of the record file depends on the number of record items. Each tuple in Att_Rel points to a record and the number of record items is determined by the node height. Specifically, the number of nodes at $h \in [1, H]$ is $O(E^{H-h})$ in which E is the node *fan-out* and $h = 1$ is for leaf nodes. Consider two extreme cases. Only one node $O(E^0)$ contains the value at the root level and $O(E^{H-1})$ nodes contain the value at $h = 1$. The number of record items is calculated by

$$\sum_{a \in dom(A)} \sum_{h \in [1,H]} E^{H-h} = \sum_{a \in dom(A)} (1 + ... + E^{H-1})$$

Theorem 3 *The storage cost is* $O(|dom(A)| \cdot H) + O(\sum_{a \in dom(A)} E^H)$.

The cost increases when the number of trajectories rises and is also proportional to the number of attribute values. We will verify this in the experimental evaluation.

4 Update

The database needs to keep track of the incoming data and allow querying both the historical and new data. An important task is to synchronize index structures in order to be consistent with the underlying data space. Given a set of incoming multi-attribute trajectories, inserting them into the index incurs updating two structures: (i) 3D R-tree and (ii) BAR. In general, a new R-tree named $\mathcal{R}_u$ is created on new trajectories and BAR is built on $\mathcal{R}_u$. To distinguish between historical and new structures, we term BAR$_u$ for the new one. $\mathcal{R}_u$ and BAR$_u$ are inserted into historical structures $\mathcal{R}$ and BAR, as illustrated in Fig. 7.

4.1 Updating R-tree

We pack the incoming trajectories and create a new R-tree by bulk load. The new R-tree is maintained by the same storage file as the historical R-tree in order to simplify the procedure of accessing the structure. Otherwise, one has to detect whether the accessed node belongs to the new R-tree or the historical R-tree and select the corresponding file. It is a rather complex task to maintain many storage files for frequent updates.

$\mathcal{R}_u$ is inserted into $\mathcal{R}$ as follows. Let H_u and H denote the heights of the two R-trees, respectively. We assume $H_u \leq H$ because the amount of incoming trajectories for one time update is usually much smaller than that of the historical data. If $H_u = H$, a new root node is created to hold root nodes of $\mathcal{R}$ and $\mathcal{R}_u$ as two entries. If $H_u < H$, we add the root node of $\mathcal{R}_u$ as an entry to an appropriate node in the target tree $\mathcal{R}$ whose height is equal to H_u. This is achieved by performing a top-down traversal in the target tree until a node whose height is equivalent to the new R-tree. During the traversal we always choose the last entry of each accessed node as the node to be processed at the next level. This is because entries are increasingly sorted by time and incoming trajectories are certainly located after existing trajectories. If the node is not full, we insert the root of the new R-tree as an entry into the node. Otherwise, we create a new node for the new R-tree.

Consider a new trajectory $mo_6 = (< u_{12}, u_{13} >, \text{(SILVER, VW)})$ in Fig. 8. We exemplify the update procedure as follows. $\mathcal{R}_u$ contains only one node and will be inserted into the 3rd entry in N_R. We record a path denoted by $\mathcal{P}_u$ along with inserting $\mathcal{R}_u$ into $\mathcal{R}$. The path starts from the root node and ends at the node to insert the entry. Each element in $\mathcal{P}_u$ is of the format (h, nid, pos) in which h is the node height, nid is the node id, and pos marks the

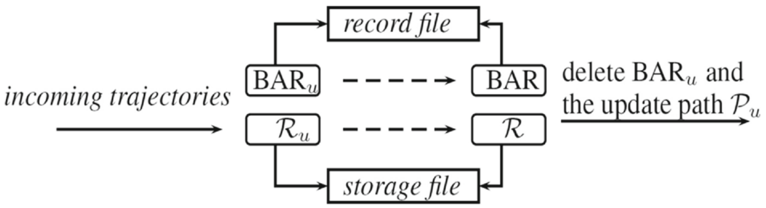


Fig. 7 An outline for updating

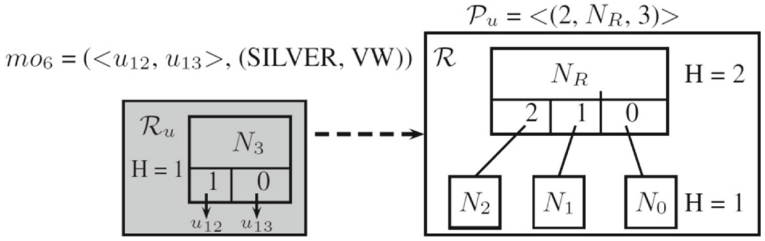


Fig. 8 Insert $\mathcal{R}_u$ into $\mathcal{R}$

entry position in the updated node. After the insertation the path will be $\mathcal{P}_u$ = <(H, *nid*, *pos*),..., (H_u, *nid*, *pos*)>, and we reversely update the spatio-temporal box of each node in the path.

Theorem 4 *the time complexity*
Inserting a new R-tree into the existing one requires $O(H)$ in which H is the height of the existing R-tree.

Proof We require $O(H)$ to find the node whose height is the same as the new R-tree because only one node is accessed at each level. Then, we update each node in the path by following the bottom-up approach. The length of the path is $O(H)$ and thus the updating cost is $O(H)$. To sum up, the merging operation requires $O(H)$. □

Theorem 5 *the number of node accesses*
Inserting a new R-tree incurs $O(H)$ node accesses in total.

Proof Finding the node in the existing R-tree to insert the new R-tree needs to access $O(H)$ nodes and only these nodes will be updated after the insertion. □

4.2 Updating BAR

The procedure of building BAR$_u$ is the same as in Section 3.3.1. To update BAR, we insert BAR$_u$ into BAR: step 1, for each tuple in BAR$_u$.Att_Rel, we search the *matching* tuple in BAR.Att_Rel and append record items for the nodes in $\mathcal{R}_u$; step 2, we update record items for each node appearing in $\mathcal{P}_u$.

Definition 7 Matching tuple
A tuple $y \in$ BAR.Att_Rel is the matching tuple of the tuple $x \in$ BAR$_u$.Att_Rel if x.A_VAL = y.A_VAL $\wedge$ $x.H = y.H$.

Figure 9 shows BAR$_u$ for the new trajectory. Consider the tuple for SILVER at $H = 1$. After determining the matching tuple, we append a new item in Record 2 for N_3. If the matching tuple is not found (the attribute value does not appear in BAR), we create the tuple as well as the record and insert them into BAR.

$\mathcal{R}_u$ is inserted into $\mathcal{R}$ as a sub-tree and the nodes in $\mathcal{P}_u$ are updated in terms of (i) spatio-temporal boxes; and (ii) attribute values. Part (i) is processed in Section 4.1 and part (ii) is processed as follows. For each attribute value in new trajectories, we look for tuples in

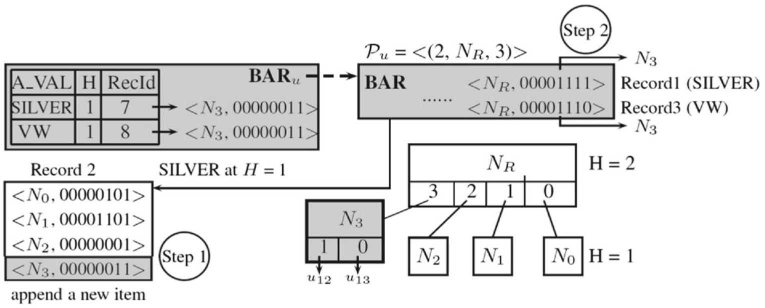


Fig. 9 Exemplify the update

BAR.Att_Rel having the value and the appropriate height according to $\mathcal{P}_u$. Note that the height is increasingly numbered from leaf to root level, guaranteeing that the heights of $\mathcal{R}$ and $\mathcal{R}_u$ are consistent. If we find the tuple, we access the record to update the item for the node. Precisely, the bitmap and the time box are updated. For example, the items of Records 1 and 3 are updated in Fig. 9 because N_3 is inserted into N_R. Later, we refresh the record to synchronize the data.

New trajectories incur an ongoing expansion of the time. The time range of the nodes in $\mathcal{P}_u$ overlaps with that of new trajectories. To enhance the update performance, we increasingly sort the items on time and update the items from the end of the list.

Definition 8 Sorted record items

The items in a record are increasingly sorted according to time intervals. Consequently, Definition 6 is updated to $D_{\underline{rec}} = \{(<nid_1, b_1, t_1), ..., (nid_n, b_n, t_n) > | t_1 < ... < t_n, nid_i \in \underline{int}, b_i \in \underline{int}, t_i \in \underline{interval}\}$

If a new root node is created, we will not find the *matching* tuple in BAR.Att_Rel. Therefore, we create the tuple as well as the record and insert them into BAR. Afterwards, we drop BAR$_u$ and $\mathcal{P}_u$. Let $\mathcal{D}_u$ be the set of new arrival trajectories. The update algorithm is given in Algorithm 2 and the complexity is provided.

Algorithm 2 *Update(BAR, BAR$_u$, $\mathcal{P}_u$, $\mathcal{D}_u$, H_u)*

```
1:  for all attribute value $a$ in $\mathcal{D}_u$ do
2:      for all $h_u \in [1, H_u]$ do                                   ▷ step 1
3:          get the tuple in BAR$_u$.Att_Rel;
4:          if find the matching tuple in BAR.Att_Rel then
5:              append items in BAR$_u$ to the record of the matching tuple;
6:          else
7:              create a tuple and insert into BAR;
8:      for all $(h, nid, pos) \in \mathcal{P}_u$ do                    ▷ step 2
9:          if $\exists\, y \in BAR.Att\_Rel$: $y$.A_VAL $= a \wedge y.H = h$ then
10:             update the record item;
11:         else
12:             create the tuple and insert into BAR;
13: delete BAR$_u$ and $\mathcal{P}_u$;
```

Theorem 6 *The time complexity of updating is* $O(|dom(A)| \cdot H)$

Proof The time complexity depends on (i) the domain of attribute values and (ii) the maximum value between H_u and $|\mathcal{P}_u|$, that is $O(H)$. □

4.3 Maintaining light-weight BAR

By performing some test workloads, we find that step 1 is not a costly procedure in terms of I/O accesses because one just appends new record items or creates tuples. On the contrary, step 2 is an expensive procedure in which we update record items for the nodes appearing in $\mathcal{P}_u$. For each attribute value, we look for the tuple in BAR that contains the value and the corresponding height. We then access the record file to load the entire record into the main-memory, find the record item to update (may also create), and write the record back into the disk. This incurs a lot of I/O accesses because all items are loaded from the disk. Figure 10 illustrates the procedure of updating the records for SILVER.

In view of the deficiency, we propose a light-weight BAR named *lw*-BAR to reduce the I/O cost. We buffer record items in *lw*-BAR rather than update BAR. A relation and a B-tree make up *lw*-BAR and there is no record file. The relation schema is of the form

$$\text{lw-Att\_Rel: } (A\_VAL : \underline{int}, H : \underline{int}, RecItem : \underline{rec}).$$

For each attribute value appearing in BAR$_u$, we use a tuple in lw-Att_Rel for the node in $\mathcal{P}_u$. In the update path, there is only one node at each height from H_u to H, and therefore the number of updated items in a record is one. The *lw*-BAR only stores one item in a record. To have a compact structure, we merge the record component (managed in a record file in BAR) into the relation by replacing the record id by the record item. Employing the *lw*-BAR, the number of I/O accesses for updating will be considerably reduced because only one tuple is processed. Continuing the example, the light-weight structure is depicted in Fig. 11.

To accommodate frequent updates, the record items for $\mathcal{P}_u$ have to be updated whenever $\mathcal{R}_u$ is inserted into $\mathcal{R}$. If the entries in the inserted node do not overflow, we only update record items for historical nodes. However, under a continuous update load, frequent insertions will cause the node overflow and lead to new nodes, as illustrated in Fig. 12. In this scenario, we merge the record item in *lw*-BAR into the one in BAR and create another record item in *lw*-BAR for the new node. We give the enhanced update algorithm in Algorithm 3.

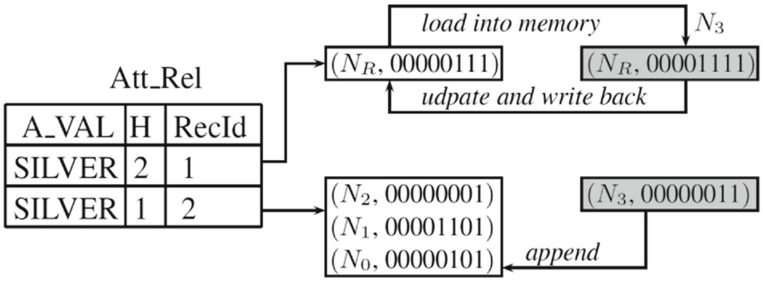
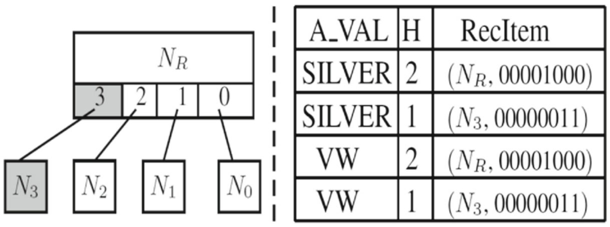


Fig. 10 Update records



A_VAL	H	RecItem
SILVER	2	(N_R, 00001000)
SILVER	1	(N_3, 00000011)
VW	2	(N_R, 00001000)
VW	1	(N_3, 00000011)

Fig. 11 *lw*-BAR for mo_6 (SILVER, VW)

Algorithm 3 *EnhanceUpdate*(BAR, BAR_u, $\mathcal{P}_u$, $\mathcal{D}_u$, H_u)

```
1: lw-BAR ← ⊘;
2: for all attribute value $a$ in $\mathcal{D}_u$ do
3:     the same as lines 2-7 in Algorithm 2;
4:     for all $(h, nid, pos) \in \mathcal{P}_u$ do
5:         if nid is not a new node then
6:             insert a tuple into lw-BAR;
7:         else
8:             merge the record item in lw-BAR into BAR and create a new item in lw-BAR
               for the new node;
9: BAR ← lw-BAR;
10: delete $\text{BAR}_u$ and $\mathcal{P}_u$;
```

Our method has low latency which is very important for on-line applications. The query evaluation can be executed when $\mathcal{R}_u$ and BAR_u are created but not yet inserted into $\mathcal{R}$ and BAR. If the system detects that the updating is not complete, we can separately search $\mathcal{R}$, BAR and $\mathcal{R}_u$, BAR_u and perform the union on their results to obtain the final answer.

4.4 The cost model

Recall step 2 in which for each attribute value we update the record for each node in $\mathcal{P}_u$. The number of updated records is determined by (i) the domain of attribute values *dom*(A), and

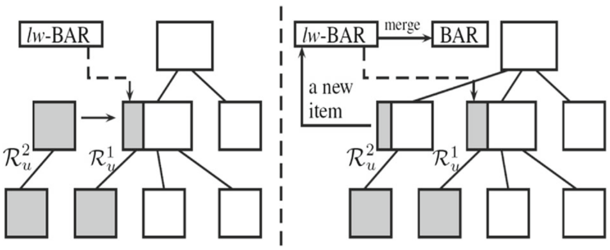


Fig. 12 Overflow and Update *lw*-BAR, BAR

(ii) the length of the update path, denoted by $|\mathcal{P}_u|$. Let *Rec* denote the I/O cost of accessing a record. We measure the update cost by

$$Cost(\text{I/O}) = \sum_{h \in [H_u, H]} \sum_{a \in dom(A)} Rec \tag{2}$$

We assume that $|dom(\text{A})|$ is static. Updating new trajectories one by one is expensive because each update involves $Cost(\text{I/O}) = H \cdot |dom(A)| \cdot Rec$. The larger H_u is, the less the cost is. An optimal case occurs if $H_u = H$. In this setting, a new root node is created and a new record is created for the root node, leading to $Cost(\text{I/O}) = |dom(A)| \cdot Rec$. This is consistent with the fact that updating by bulk load is more efficient than individually performing the update [4]. *Rec* is the cost of loading record items for all nodes containing an attribute value at a particular height. E is the capacity of the R-tree node. Let B be the block size. The I/O cost of accessing a record for the nodes at h is estimated by

$$Rec = \frac{E^{H-h}}{B}, h \in [H_u, H] \tag{3}$$

Combining Eqs. 2 and 3, the update cost is

$$Cost(\text{I/O}) = \sum_{h \in [H_u, H]} |dom(A)| \cdot \frac{E^{H-h}}{B}. \tag{4}$$

Employing the *lw*-BAR, we only insert one record item for each attribute value at each height, reducing the cost to $O(\frac{1}{B})$. In contrast, the method in Section 4.2 has to load all record items into the main-memory. If a new node is created, record items for all nodes at H_u are loaded into the main-memory, leading to $O\left(\frac{E^{H-H_u}}{B}\right)$, although only one item is updated. To sum up, the cost of the enhanced method is

$$Cost(\text{I/O})_{lw} = \frac{E^{H-H_u}}{B} + \sum_{h \in [H_u, H]} \frac{|dom(A)|}{B} \tag{5}$$

5 The algorithm for C*k*NN_MAT

We outline the query procedure in Fig. 13. By default, we introduce the method by considering a single value for each attribute and will explicitly point out the difference for multiple values when necessary.

Filter We access BAR to determine the R-tree nodes that include query attribute values and intersect the query time, denoted by N_a. We take N_a coupled with mq and k as input to traverse the R-Tree in breadth-first order, during which the search space is pruned by taking into account spatial and temporal parameters as well as attributes. The filter returns a set

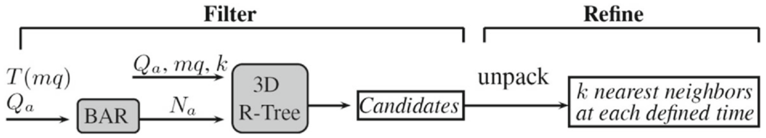


Fig. 13 The query framework

of candidates, each of which fulfills the attribute condition and approximately belongs to k nearest neighbors.

Refinement Unpack each candidate trajectory to get the original temporal units and perform the exact distance computation to return k nearest neighbors at each query time.

Since the refinement is trivial, we focus on the filter phase, as given in Algorithm 4, which calls two subroutines *CollectNodes* and *Candidates*, respectively.

Algorithm 4 *Filter*(Q_a, mq, k, *BAR*, $\mathcal{R}$, $\mathcal{D}$)

```
1: let $T(mq)$ return the query time;
2: $N_a \leftarrow$ CollectNodes($Q_a$, $T(mq)$, BAR);
3: return Candidates($N_a$, $Q_a$, $mq$, $k$, $\mathcal{R}$, $\mathcal{D}$);
```

5.1 Collecting R-tree nodes

We collect the nodes containing Q_a level by level. For each $a \in Q_a$, we start from $h = 1$ and access BAR to find the records. For each item (nid, b, t) in the record, we check whether the item identified by nid is already in N_a. If not, we insert the item into N_a by attaching a *counter*, initialized by 1. Such a value represents the number of query attribute values contained in the node. The extended record item is denoted by λ. If the item already exists in N_a, we increase the *counter* and update the bitmap by performing the bitwise AND. This is because a node fulfilling the condition must contain all values in Q_a. Let $|Q_a|$ return the number of attribute values. We present the algorithm in Algorithm 5 and use Lemma 3 to prune the items. Figure 14 depicts the procedure of determining the nodes for Q_a = (SILVER, VW).

Algorithm 5 *CollectNodes*(Q_a, $T(mq)$, *BAR*)

```
1: $N_a \leftarrow \emptyset$;
2: for all $a \in Q_a$ do
3:     for all $h \in [1, H]$ do
4:         select the tuple from BAR.Att_Rel and retrieve the record RD;
5:         for all ($nid$, $b$, $t$) $\in RD$ do
6:             if $\exists\, \lambda \in N_a$: $\lambda.nid = nid$ then
7:                 if $\lambda.t \cap t$ then
8:                     $\lambda.counter$++;
9:                     $\lambda.b \leftarrow \lambda.b$ AND $b$;
10:            else
11:                if $t \cap T(mq)$ then
12:                    $\lambda \leftarrow (nid, b, t, counter)$;
13:                    $N_a \leftarrow \lambda$;
14: for all $\lambda \in N_a$ do
15:     remove $\lambda$ according to Lemma 3;
16: return $N_a$;
```

Lemma 3 *Given an item $\lambda \in N_a$, the item is pruned if $\lambda.counter \neq |Q_a|$ or $\lambda.b = 0$.*

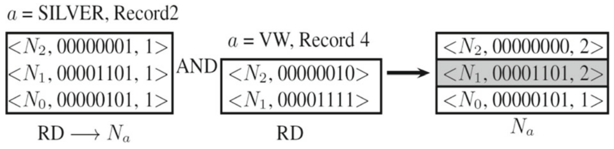


Fig. 14 Determine the nodes at $h = 1$

Proof (*i*) $\lambda.counter \neq |Q_a|$: It is impossible that $\lambda.counter > |Q_a|$ because distinct attribute values are counted. If $\lambda.counter < |Q_a|$, the number of attributes in the node is less than $|Q_a|$ and λ can be safely pruned.

(*ii*) $\lambda.b = 0$: There is no entry in the node containing all query values and we safely prune λ.

□

Time complexity The time cost includes (i) searching BAR.Att_Rel to find the tuples and (ii) collecting record items. We perform a lookup on the B-tree for each attribute value at each $h \in [1, H]$. This requires

$$O(d \cdot H \cdot \log(|dom(A)| \cdot H))$$

as at most d attributes are queried. Next, we consider the number of processed record items for an attribute value at a particular height. Such a value is equivalent to the number of nodes fulfilling the attribute condition. Obviously, the maximum value occurs at leaf level, leading to $O(E^{H-1})$ nodes. Assume that attribute values are uniformly distributed. Let

$$\delta_a = \max\left\{\frac{1}{|dom(A_1)|}, ..., \frac{1}{|dom(A_d)|}\right\}$$

be the maximum ratio of a single value to all values in the domain among all attributes and δ_t be the ratio of $T(mq)$ to the overall time. The number of leaf nodes containing a query value is approximated by

$$O(E^{H-1} \cdot \delta_a \cdot \delta_t).$$

The total number of accessed record items will be

$$\sum_{h\in[1,H]} E^{H-h} \cdot \delta_a \cdot \delta_t = O(E^H \cdot \delta_a \cdot \delta_t).$$

Theorem 7 *Given a query with $O(d)$ attribute values, the time to report the nodes containing the query is $O(d \cdot (H \cdot \log(|dom(A)| \cdot H) + E^H \cdot \delta_a \cdot \delta_t))$.*

5.2 An extension: multiple values

We make an extension that a query attribute defines multiple values. A node is satisfied if it contains one of the values. We extend the aforementioned $\lambda \in N_a$ to (nid, b, t, aid) by adding the attribute id. The bitwise OR is performed on bitmaps for values from the same attribute. Algorithm 5 is slightly modified, see Algorithm 5.2, and Lemma 4 is used for pruning.

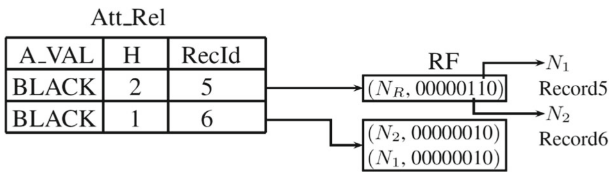


Fig. 15 BLACK in BAR

Algorithm 6 *CollectNodes2*$(Q_a, T(mq), BAR)$

```
1:  lines 1-4 in Algorithm 5;
2:  for all $(nid, b, t, aid) \in RD$ do
3:      if $\exists \lambda \in N_a$: $\lambda.nid = nid$ then
4:          if $aid \neq \lambda.aid$ then
5:              if $t \cap \lambda.t$ then
6:                  $\lambda.counter{+}{+}$;
7:                  $\lambda.b \leftarrow \lambda.b$ AND $b$;
8:          else
9:              if $t \cap T(mq)$ then
10:                 $\lambda.b \leftarrow \lambda.b$ OR $b$;
11: lines 10-12 in Algorithm 5;
12: for all $\lambda \in N_a$ do
13:     remove $\lambda$ using Lemma 4;
14: return $N_a$;
```

Lemma 4 *Let AttCount*(Q_a) $(\leq |Q_a|)$ *return the number of query attributes. An item* $\lambda \in N_a$ *is pruned if* $\lambda.counter \neq AttCount(Q_a)$ *or* $\lambda.b = 0$.

Consider the query "Report the nearest SILVER or BLACK VW to mo_4". We look for qualified nodes that contain SILVER/BLACK and VW. Based on Fig. 5, we report BAR for BLACK by referring to Fig. 15.

We exemplify the procedure of reporting qualified nodes for ({SILVER, BLACK}, VW) at $h = 1$ in Fig. 16. Records 2 and 6 are returned for SILVER and BALCK, respectively. We perform the bitwise OR on bitmaps and do not increase the *counter*. Next, we find the nodes containing VW and perform the bitwise AND to determine the nodes containing {SILVER,

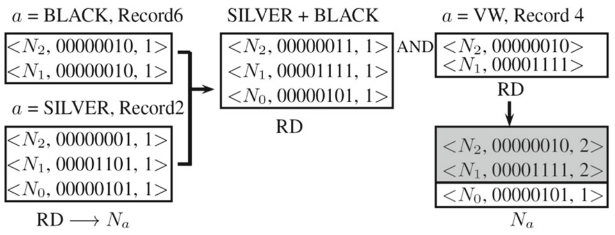


Fig. 16 Multiple attribute values at $h = 1$

BLACK} and VW. The counter increases if the bitmap is not zero. Non-qualified nodes are pruned according to Lemma 4.

Theorem 8 *The time cost for multiple values is* $O(|dom(A)| \cdot (H \cdot \log(|dom(A)| \cdot H) + E^H \cdot \delta_a \cdot \delta_t))$.

5.3 Reporting candidates

The algorithm, given in Algorithm 7, traverses the R-tree from root to leaf level during which we prune the search space using both spatio-temporal and attribute predicates. A list $\mathcal{L}$ is used to maintain accessed nodes. For each node $n \in \mathcal{L}$, we determine whether (i) n belongs to N_a; and (ii) objects in the subtree rooted at n will contribute to the result. If $n \notin N_a$, we prune it. If $n \in N_a$, we extend the method in [21] to maintain the candidates in terms of count and their distances to mq. The query needs only k neighbors. If there are k candidates at each defined time, objects that are further than current candidates to mq can be safely pruned, see Lemma 5.

Algorithm 7 *Candidates*$(N_a, Q_a, mq, k, \mathcal{R}, \mathcal{D})$

```
1: $\mathcal{C} \leftarrow \oslash$;
2: $\mathcal{L} \leftarrow \mathcal{R}.RootId$;
3: while $\mathcal{L}$ is not empty do
4:     $n \leftarrow SelectNode(\mathcal{R}, \mathcal{L}.top())$;
5:     if $\exists\ \lambda \in N_a : \mathcal{L}.top() = \lambda.nid \wedge$ candidates are not enough then        ▷ Lemma 5
6:         $P \leftarrow \oslash$;
7:         if $|\lambda.b| = m$ then
8:             perform a linear search to set $P$;
9:         else
10:            $P \leftarrow ReportBitPos(\lambda.b)$;
11:        for all entry $e \in n$ according to $P$ do
12:            if $n$ is a leaf node then
13:                $mo \leftarrow SelectTuple(\mathcal{D}, e.pointer)$;
14:                if $T(e) \cap T(mq) \wedge$ contain$(mo.Att, Q_a)$ then
15:                    $\mathcal{C} \leftarrow mo.Trip$;
16:            else
17:                $\mathcal{L} \leftarrow e.pointer$;
18: return $\mathcal{C}$
```

Lemma 5 *Given a node n, if there are k objects in the candidate set during* $T(n)$ *and their distances to mq are smaller than dist(n, mq) at each* $t \in T(n)$, *n is safely pruned.*

To determine whether there are enough candidates, a segment tree is maintained by storing the time interval, the distance and the number of trajectories in a node. The key point is that the number of trajectories in the segment tree is not the number of all trajectories in an R-tree node, but the number of trajectories fulfilling the attribute condition. Such a value depends on the query. Although we can do the pre-computation for each attribute value and use it for $|Q_a| = 1$, one has to calculate on-the-fly for $|Q_a| > 1$, which is a costly procedure.

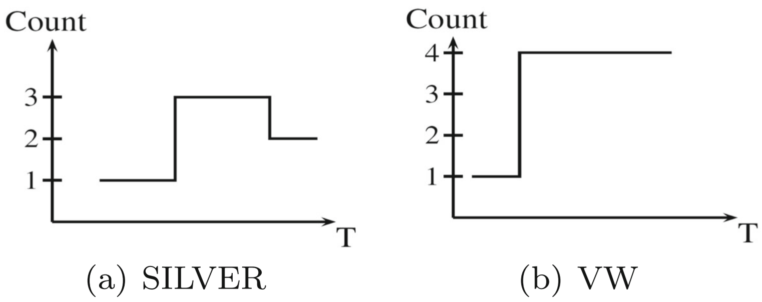


Fig. 17 The number of trajectories in N_R for SILVER and VW

This is because the result is the intersection set of several query attributes and one needs to traverse the sub-tree to count the number of trajectories containing query attributes. We do the pre-computation for a single value. For each $a \in dom(A)$, we maintain a temporal integer to represent the distribution. Each node needs $O(|dom(A)|)$ temporal integers. If $|Q_a|$ = 1, we create candidates using temporal integers and insert them into the segment tree. If $|Q_a| > 1$, we receive temporal integers until trajectories (leaf nodes) are accessed. Using the index in Fig. 5, we report possible temporal integers for SILVER and VW of the root node, as illustrated Fig. 17. We do the intersection on SILVER and VW to determine trajectories containing both attributes and report the trajectory count.

If n belongs to N_a and candidates are not enough, we open the node to look up entries and search objects. We call *ReportBitPos* to establish the entries containing query values. If n is a leaf node, we retrieve the tuple from the relation to precisely evaluate the attribute. This is because if a bit maps to several entries ($m < E$), we set the bit as long as one entry contains the value. Therefore, the value is accurately checked for each entry. If $m \geq E$, this step is ignored. If n is a non-leaf node, its child nodes will be put into $\mathcal{L}$ for further consideration. Using the running example, Fig. 18 exemplifies the procedure of reporting candidates.

The procedure of processing multiple query values is the same as processing a single value, but more nodes will be accessed. Using the query "Continuously report the nearest SILVER or BLACK VW to mo_4", we depict the procedure of reporting candidates in Fig. 19.

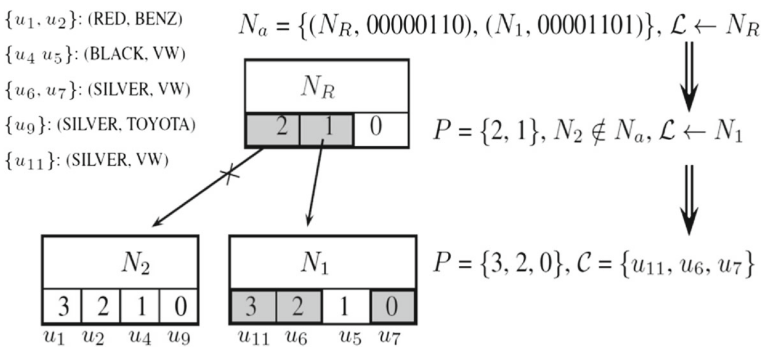
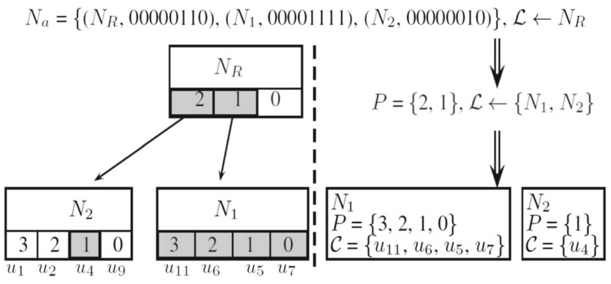


Fig. 18 Return candidates for (SILVER, VW)



$N_a = \{(N_R, 00000110), (N_1, 00001111), (N_2, 00000010)\}, \mathcal{L} \leftarrow N_R$
N_R
2 1 0
$P = \{2, 1\}, \mathcal{L} \leftarrow \{N_1, N_2\}$
N_2
3 2 1 0
u_1 u_2 u_4 u_9
N_1
3 2 1 0
u_{11} u_6 u_5 u_7
N_1 $P = \{3, 2, 1, 0\}$ $\mathcal{C} = \{u_{11}, u_6, u_5, u_7\}$
N_2 $P = \{1\}$ $\mathcal{C} = \{u_4\}$

Fig. 19 Candidates for ({SILVER, BLACK}, VW)

5.4 The refinement

This step includes two phases: (i) unpack each candidate to obtain temporal units; (ii) apply the slightly modified plane-sweep algorithm [3] to determine the k lowest time-dependent distance curves to report the result.

Phase (i) is straightforward. Phase (ii) takes in a sequence of candidate trajectories, temporally ordered on time. We compute their time-dependent distances to the query trajectory and find the k nearest objects at each defined time point. To achieve this, one needs to determine pieces of movements with overlapping time and apply the distance function by employing the linear interpolation on each piece [15, 16]. We manipulate temporal units to calculate the distance. The distance function is of the form the square root of a quadratic polynomial, as illustrated in Fig. 20. The lowest curve corresponds to the nearest trajectory to the query. Since distances change over time, *split points* between curves are found to determine the k lowest pieces of curves.

6 Experimental evaluation

We implement the proposal in C/C++ and perform the evaluation in an extensible database system SECONDO [20]. A standard PC (Intel(R) Core(TM) i7-4770CPU, 3.4GHz, 4GB memory, 2TB hard disk) running Suse Linux 13.1 (32 bits, kernel version 3.11.6) is used.

6.1 Datasets and parameter settings

Real datasets are from a company Datatang including Shanghai taxis (TAXI) and Beijing buses (BUS) [1]. TAXI includes taxi GPS records from four companies in 2014. The

Fig. 20 Distance curve

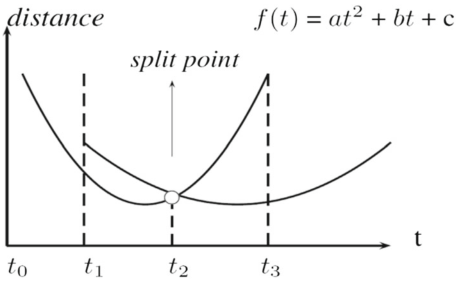


Table 3 Datasets and parameters

Name	$\|\mathcal{D}\|$ (million)	$\|U\|$ (million)	d	$dom(A)$
TAXI	3.0	32.5	1	[1, 4]
BUS	1.1	8.3	1	[1, 384]
MAT5	1.8	110.88	10	[1, 127]
Parameter settings				
$\|Q_a\|$	$\{1, 2, \underline{3}, 4, 5\}$		k	$\{1, \underline{5}, 10, 20, 50, 100\}$

company id is defined as the attribute. BUS contains bus card records in 2014. Each record stores when (the time reading the card) and where (locations of bus stops) passengers get on and off the bus. Each bus is identified by its id and bus stops are identified by orders in the route and their locations. We build bus trips from bus card records by grouping records on bus id and then sorting them on time. There are 384 bus routes in total and the route id is set as the attribute. Part of the data is published at (http://dbgroup.nuaa.edu.cn/jianqiu/).

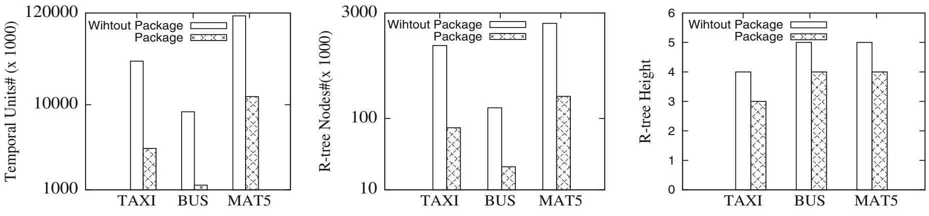
Synthetic datasets are generated by utilizing a tool MWGen [56]. Several datasets are created in terms of the number of trajectories, the attribute quantity and the attribute domain. For each attribute, the value is randomly selected from the domain. In the system, one can flexibly define parameters to generate datasets in different settings. We provide the default dataset MAT5 here and will report other datasets when they are used in the evaluation. In each query, the standard trajectory and attribute values are randomly selected over domains. Datasets statistics and query parameters are listed in Table 3, in which default query values are marked.

6.2 The effect of packing

We report the packing effect on all datasets by referring to Fig. 21. A compact dataset is achieved and the number of temporal units decreases by an order of magnitude, demonstrating that the effect in practice is better than the theoretical analysis in Lemma 1. In turn, the number of R-tree nodes is greatly reduced and the tree height decreases. This leads to an index structure with small size, reducing the storage overhead and accessing cost.

6.3 Searching the bitmap

We compare the performance of querying the bitmap between the binary search and the linear search. The results are reported in Fig. 22. The binary search achieves better



(a) Temporal units (b) R-tree nodes (c) R-tree heights

Fig. 21 Compact trajectory approximation

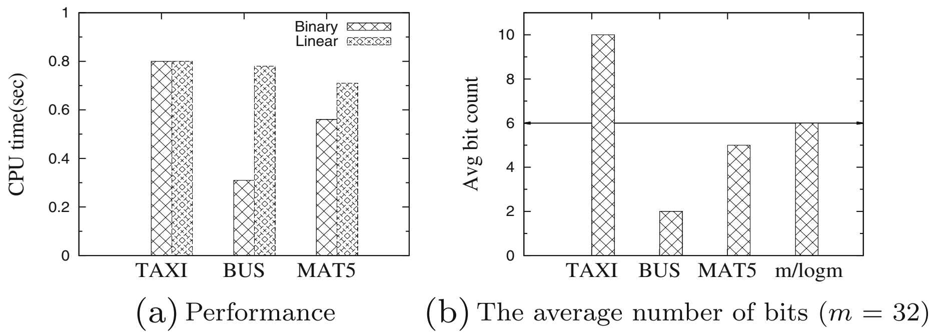


Fig. 22 Querying the bitmap ($|Q_a|$ = 3 and k = 5)

performance than the linear search for BUS and MAT5, but the two methods have the same performance for TAXI. This is because the average number of bits per attribute value is less than $O(m \log m)$ for BUS and MAT5, and therefore the binary search guarantees an optimal performance, as we analyzed in Section 3.3.3. However, this does not hold for TAXI whose attribute domain is small ($dom(A)$ = [1, 4]) and the average number of bits per attribute value is larger than $m \log m$.

6.4 Performance evaluation

In the evaluation, CPU time and I/O accesses are used as performance metrics and the results are averaged over 20 runs. Three baseline methods are developed.

(i) **4D R-tree**. One can treat attributes as an extra dimension in addition to spatial and temporal. Therefore, a high-dimensional index can be utilized. For each multi-attribute trajectory, attribute values (a_1, ..., a_d) are decomposed into d individual values and each is combined with the standard trajectory, e.g., mo_1(RED, BENZ) will yield mo'_1(RED) and mo''_1(BENZ). A relation with the schema (Id: $\underline{int}$, $Trip$: $\underline{mpoint}$, Att: $\underline{att}$) is created in which Att is a single-value attribute. The tuple quantity of the relation is $O(d \cdot |D|)$. To answer CkNN_MAT, we need $|Q_a|$ query boxes each of which is created on the query attribute value and the defined time $T(mq)$. The procedure starts from the root node and evaluates each accessed node. If the node intersects query boxes, we open the node and access its child nodes for further consideration. Otherwise, we safely prune the node. A multi-attribute trajectory is returned only if among its d trajectories there are $|Q_a|$ trajectories intersecting query boxes. The method enlarges the data set d times, leading to high storage cost, and does not achieve a good locality when grouping objects for a large d.

(ii) **3D R-tree + Attribute Relation** (3D RAR). The paper [21] develops an algorithm that employs a 3D R-tree with auxiliary structures to efficiently answer continuous nearest neighbor queries over standard trajectories. We extend the paradigm to support proposed queries by recording the set of attribute values contained in each node. A relation stores attribute values contained in R-tree nodes. During the query procedure, we first determine whether the accessed node contains all values in Q_a by performing a lookup on a B-tree built on the relation. If yes, we open the node and move forward to the spatio-temporal evaluation. Otherwise, we prune the node. The main drawback is that recording attribute values is able to determine whether the node contains a particular value but cannot make the decision on the AND predicate for several attributes.

(iii) **3D R-tree + Inverted Bitmap** (RIB). In the field of spatial keywords search, a lot of R-tree based indexes have been proposed. According to [7], the method in [53] is the most efficient to answer boolean kNN queries. The approach partitions the data into multiple groups such that each group shares as few attributes as possible. In the context of multi-attribute trajectories, trajectories are grouped on attributes (a_1,..., a_d), which can be viewed as a point in d-dimensional space. We apply Z-order to map the d-dimensional value to one-dimensional and group objects based on Z-order values. Objects are then sorted by attribute values, time and spatial data. The 3D R-tree is employed. Each R-tree node contains a pointer to an inverted bitmap that records positions of entries defining the attribute value. A relation is used to store the bitmaps and a B-tree is created by combining the node id and the attribute value as the key. The bitmap length is the *fan-out* of a node. For each accessed node, a sequential scan is made on the bitmap to determine entries containing query values.

6.4.1 Scalability

We conduct the experiments to obtain insights into different aspects affecting the performance: (i) the number of multi-attribute trajectories; and (ii) attributes. Table 4 reports dataset settings.

Table 4 Datasets for scalability

Name	$\|\mathcal{D}\|$ (mil)		
(a) $\|\mathcal{D}\|$, d = 10			
MAT1	0.27		
MAT2	0.54		
MAT3	0.9		
MAT4	1.35		
MAT5	1.8		
d	$dom(A)$		
(b) scaling d			
1	[1, 5]		
3	[1, 28]		
6	[1, 76]		
10	[1, 127]		
20	[1, 247]		
d	$dom(A)$	d	$dom(A)$
(c) scaling $dom(A)$			
1	[1, 5]	10	[1, 79]
	[1, 13]		[1, 127]
	[1, 50]		[1, 247]
	[1, 100]		[1, 427]
	[1, 150]		[1, 857]

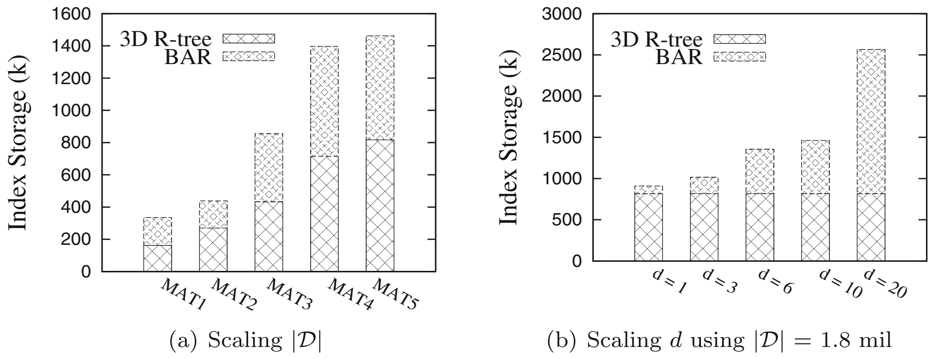


Fig. 23 Index storage

Index storage When scaling $|\mathcal{D}|$ and d, we report the storage cost of the index structure including 3D R-tree and BAR. As shown in Fig. 23, the storage costs of both structures increase when we enlarge the number of trajectories. By fixing the number of trajectories, the storage cost of BAR rises proportionally when d is enlarged. This is consistent with the analysis in Section 3.3.4.

Scaling $|\mathcal{D}|$ The query efficiency is reported in Fig. 24. The results demonstrate that when the data size grows the costs of all methods rise proportionally, but our method achieves two orders of magnitude better performance than baseline methods.

Tuning d ($|\mathcal{D}|$ = 1.8 million) Figure 25 reports the result of tuning d in which $|Q_a| = 3$ by default and $|Q_a| = 1$ for $d = 1$. Our method achieves the best performance in all settings and RIB performs competitively to our method when $d = 3$. RIB orders the data on attribute values and the predicate has a good selectivity when $|Q_a| = d$. However, when d increases, the performance degrades dramatically. We analyze that Z-order values cannot preserve the locality when the dimension becomes large and the linear scan is inferior to our bitmap querying approach. The 4D R-tree has poor performance because the dataset is enlarged d times.

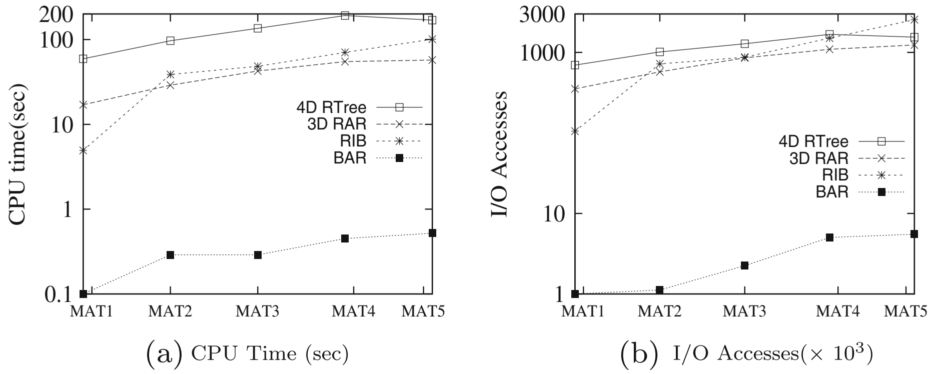


Fig. 24 Scaling $|\mathcal{D}|$

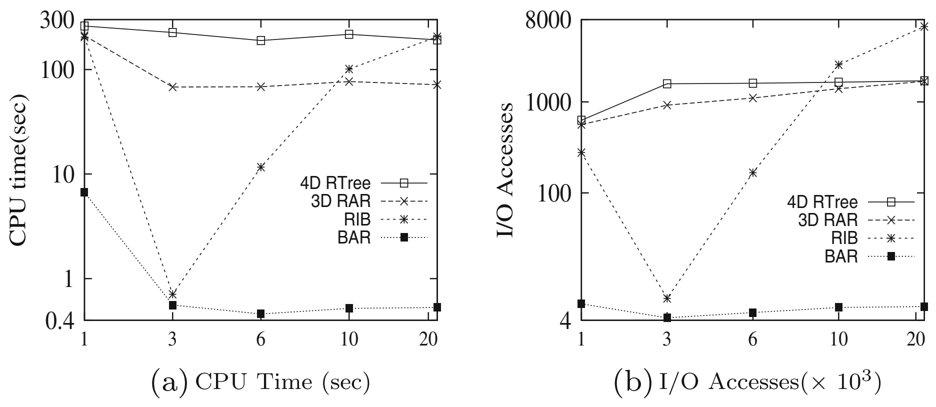


Fig. 25 Scaling d

Tuning *dom*(A) ($|\mathcal{D}|$ = 1.8 million) We enlarge attribute domains by conducting two experiments: $d = 1$ and $d = 10$. In the former case, there is no AND predicate ($|Q_a| = 1$) and the number of attribute values affects the attribute selectivity. In the latter case, we set $|Q_a|$ = 3. Our method consistently outperforms baseline methods in two settings, as demonstrated in Figs. 26 and 27. One can see that the performance of our method and RIB increases when $dom(A)$ becomes large. This is because the selectivity of the attribute predicate increases. The RIB performance shows different behavior by tunning d and $dom(A)$. The method is sensitive to attributes and thus limited in scope.

6.4.2 *Effect by* $|Q_a|$

Figure 28 reports the experimental results, demonstrating that our method achieves more than an order of magnitude faster than three baseline methods. The proposed method is able to efficiently find R-tree nodes containing query attribute values. When $|Q_a|$ increases, less nodes fulfill the condition, reducing the number of disk accesses. Formally, given a query attribute value $a \in dom(A_i)$, assuming $|dom(A_i)|$ is the minimum among all domains, the number of trajectories containing a is calculated by $|\mathcal{D}| \cdot \delta_a$ (defined in Section 5.1). The number of trajectories containing Q_a (a_1 AND a_2 ... $a_{|Q_a|}$) is approximated by $|\mathcal{D}| \cdot \delta_a^{|Q_a|}$.

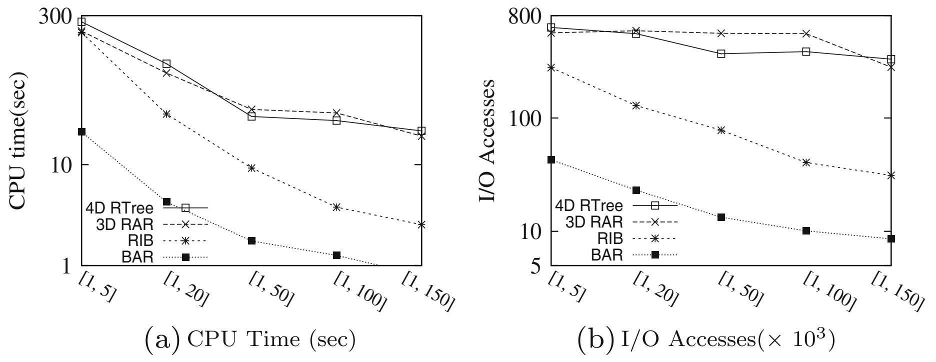


Fig. 26 Scaling $dom(A)$ (d = 1, $|Q_a|$ = 1)

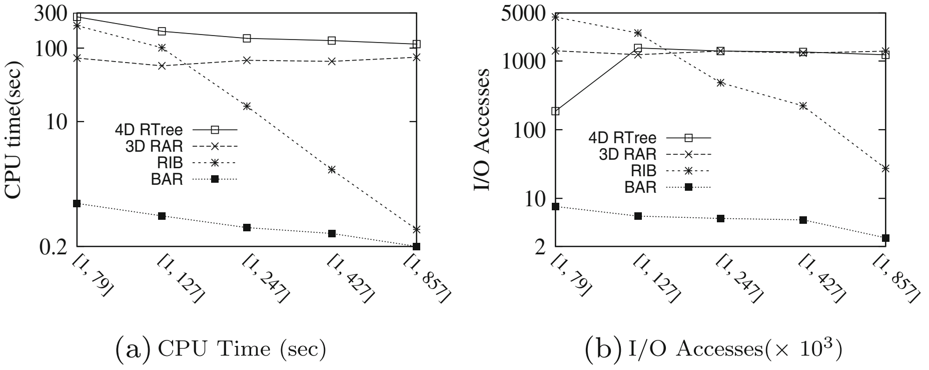


Fig. 27 Scaling $dom(A)$ (d = 10, $|Q_a|$ = 3)

6.4.3 *Effect by k*

The results are reported in Figs. 29, 30 and 31. TAXI and BUS contain only one attribute, and thus we set $|Q_a| = 1$. The query costs of all methods are not sensitive to k and our method substantially outperforms alternative methods. For BUS (d = 1, $dom(A)$ = [1, 384]), the RIB performance is close to ours in a few cases due to good attribute selectivity. However, the performance decreases for TAXI (d = 1, $dom(A)$ = [1, 4]) and MAT5 (d = 10, $dom(A)$ = [1, 127]). An interesting behavior is that the query performance is not sensitive to k. The reason is, the majority of the time cost is in the filter step in which the pruning ability is not significantly affected by k. The refinement step requires little time cost and the number of processed candidates does not rise proportionally to k.

In the filter step, we prune a node if there are enough candidates. That means, during the time covered by the node there are at least k trajectories in the candidate set whose distances to the query trajectory are smaller than the distance between the node and the query trajectory. However, for CkNN_MAT queries such a value can not be determined before the query is issued ($|Q_a| > 1$). Computing the value on-the-fly is not acceptable because we need to traverse the subtree to count all trajectories containing Q_a. We will obtain the value only

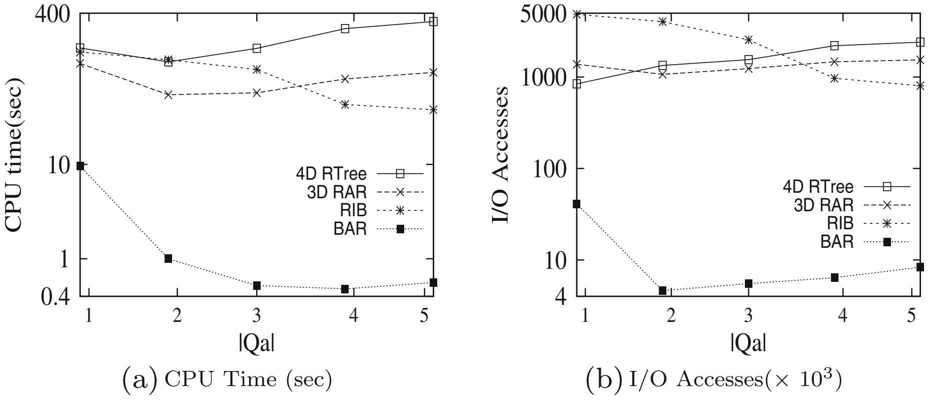


Fig. 28 Scaling $|Q_a|$ using MAT5

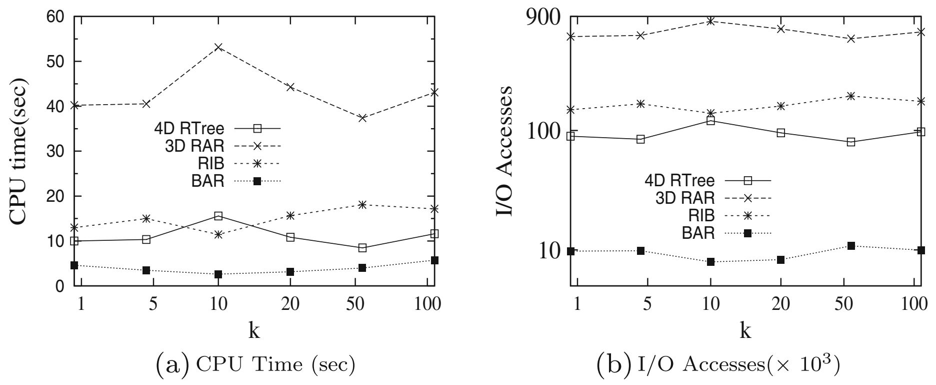


Fig. 29 Scaling *k* using TAXI

when leaf nodes are opened to evaluate trajectory attributes. The pruning method can only be applied when trajectories in leaf nodes are accessed, postponing the pruning procedure. A large number of nodes have been accessed before we can perform the pruning. We record the time cost of the filter step which is almost equal to the overall running time. Although the refinement step iteratively processes each candidate, the overall number of candidates is not large, as reported in Fig. 32. The number of candidates does not rise proportionally to k because the candidates depend on not only k but also Q_a and the query time.

6.4.4 *Applying packing to baseline methods*

Our packing method can be applied to all baseline methods. Using datasets MAT5, TAXI and BUS, we run baseline methods by building the index on packed data. Default query parameters are used. As expected, the performance of all baseline methods increases due to a compact data representation and less index overhead. Using TAXI and MAT5, our method still exhibits the best performance, about 3 times faster than other methods. The performance difference among all methods is marginal for BUS because the attribute is quite selective (Fig. 33).

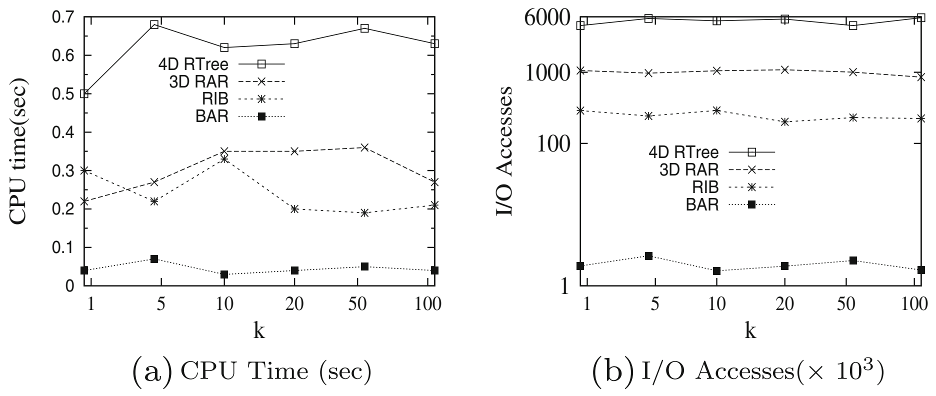
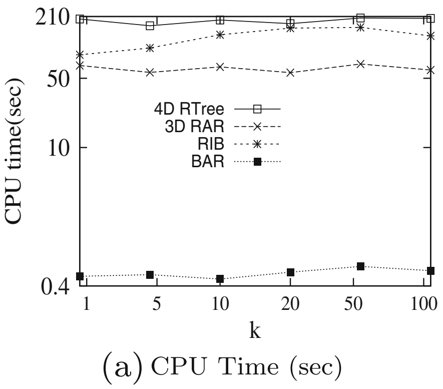


Fig. 30 Scaling *k* using BUS



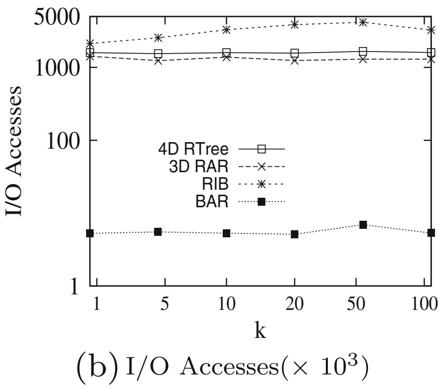


Fig. 31 Scaling k using MAT5

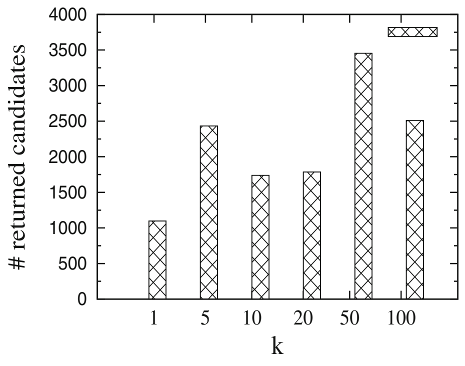
Short, median and long query trips For each standard trajectory, we calculate the trajectory deviation by considering spatial and temporal dimensions, applying Eq. 1 in Section 3.2. Then, we intentionally select *short*, *median* and *long* trajectories as query trajectories in terms of deviation values. Figure 34 shows that our method outperforms baseline methods by a factor of 3-100.

Multiple values for one attribute Using $|Q_a| = 3$, we let one of query attributes include a set of values, i.e., $|Q_a.X_i| > 1$, and scale the number of values from {2, 4, 6, 8, 10}. The other two attributes only define one value. As reported in Fig. 35, when $|Q_a.X_i|$ increases, the query costs rise for all methods because more data objects are involved in the evaluation. Our method performs at least 2 times faster than other methods.

6.4.5 Update evaluation

Update efficiency Using MAT5 and BUS, we build the historical database on part of the dataset and take the rest as new trajectories, denoted by $\mathcal{D}_u$. The performance is evaluated by scaling $|\mathcal{D}_u|$. The overall time is measured, as reported in Fig. 36a. Although $|\mathcal{D}_u|$ increases in several orders of magnitude, the updating cost only rises marginally. We also carry out the evaluation of performing a series of updates by defining $|\mathcal{D}_u|$ = 500,000 for MAT5 and $|\mathcal{D}_u|$ = 300,000 for BUS, respectively, and building the historical index on $\mathcal{D}$ - $\mathcal{D}_u$. Then,

Fig. 32 The number of returned candidates (MAT5, $|Q_a| = 3$)



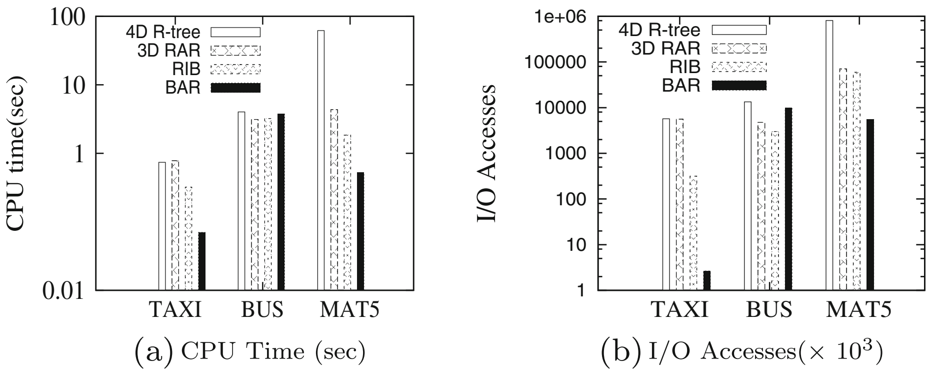


Fig. 33 Generalize packing

50 updates are continuously performed in which $\frac{|\mathcal{D}_u|}{50}$ trajectories are processed each time. We measure the overall update time and report the result in an accumulated way, as shown in Fig. 36b. The time cost increases slightly.

Performance between BAR and lw-BAR We evaluate the efficiency of updating BAR by comparing the baseline method and the one using the light-weight structure. As illustrated in Fig. 37, the lw-BAR substantially outperforms the baseline method, validating the proposed cost model. Using MAT5, an interesting behavior occurs when $|\mathcal{D}_u|$ arrives at 200,000. The two methods almost have the same I/O accesses. We detect that under this workload the height of the new R-tree is equal to that of the historical R-tree. A new root node is created to hold two subtrees and the length of the updating path is 0. A small update cost is required for both methods.

We evaluate the query performance by building the index on the same dataset in different ways. One method involves updating by initially creating the structure on $\mathcal{D}$ - $\mathcal{D}_u$ ($|\mathcal{D}_u|$ = 500,000 for MAT5 and $|\mathcal{D}_u| = 300,000$ for BUS) and then continuously performing 10 updates ($\frac{|\mathcal{D}_u|}{10}$ in each update). The other method creates the index on $\mathcal{D}$, i.e., without any update. We aim to test whether the update will influence the query performance. Default query parameters are used. Figure 38 reports the query costs, showing that continuously

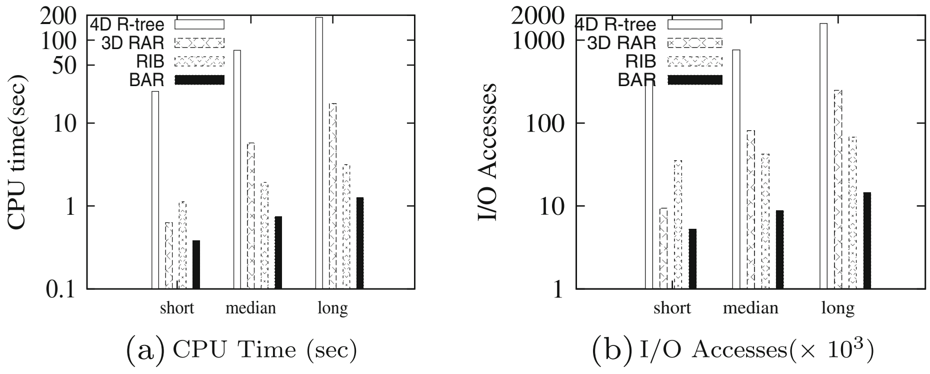


Fig. 34 Short, median and long query trajectories

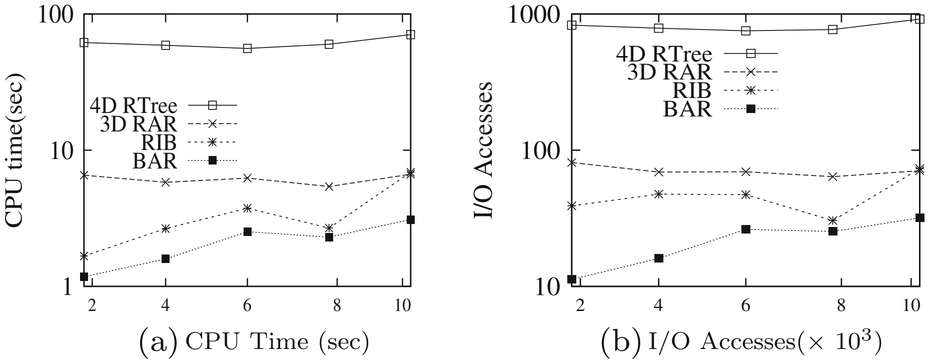


Fig. 35 Multiple query values

updating the structure will not greatly affect the performance. In fact, the performance in the update scenario is superior because a better partition is achieved in the time dimension.

6.5 Discussion

We analyze each alternative method and explain why they are inferior than our method. In order to build the 4D R- tree, an multi-attribute trajectory is decomposed into d trajectories, each of which contains a single-value attribute. This enlarges the data set d times. During the query processing, the attribute is approximately evaluated. Using Q_a = (SILVER, VW), we search the index to look for trajectories containing either SILVER or VW because each trajectory contains only one attribute. However, among returned trajectories only some contain both of them. Multi-attribute trajectories such as (SILVER, TOYOTA) or (BLACK, VW) will also be included, increasing the number of processed objects. To exactly determine the objects, we traverse the tree down until the leaf level and retrieve tuples for accurate examination.

The method using 3D RAR is able to select R-tree nodes containing an individual attribute value, but cannot determine trajectories containing several values from different attributes. Given a set of values, trajectories containing one of them will be all included

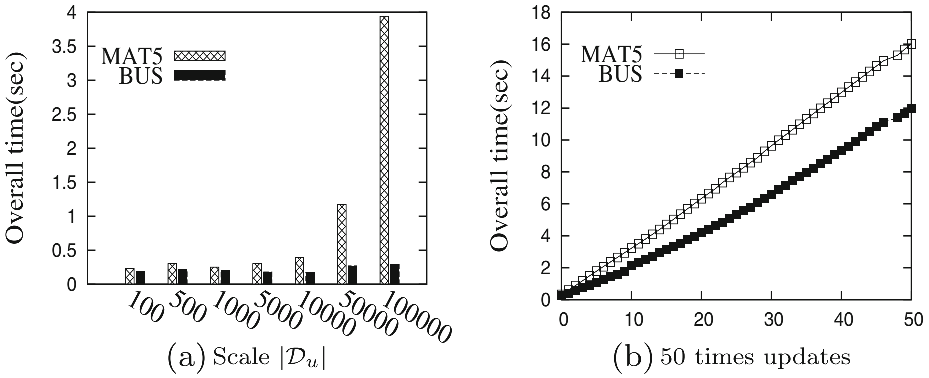


Fig. 36 Update performance

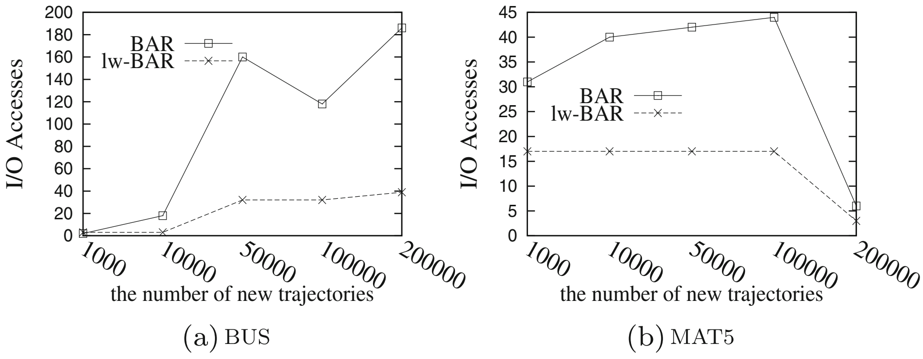


Fig. 37 BAR and lw-BAR

although some of them may not fulfill the condition. The AND predicate cannot be not evaluated before leaf nodes are accessed, weakening the pruning ability.

RIB is limited in scope because it only achieves good performance when the attribute predicate is selective, e.g., (i) $|Q_a| = d$, or (ii) $|Q_a| = 1$ and $dom(A)$ is large. In other settings, the performance significantly deteriorates. Furthermore, if traditional nearest neighbor queries are processed, the efficiency deteriorates as the spatio-temporal proximity is not preserved. Our method achieves stable performance and generalizes to queries on standard trajectories.

7 Generality

The generality of our method includes three aspects: (i) packing standard trajectories, (ii) managing attribute values, and (iii) supporting a range of queries on multi-attribute trajectories and also queries on standard trajectories.

Packing The established method produces a compact data set. There is no information loss and no extra storage cost. The procedure can be applied for other trajectory queries to enhance the performance.

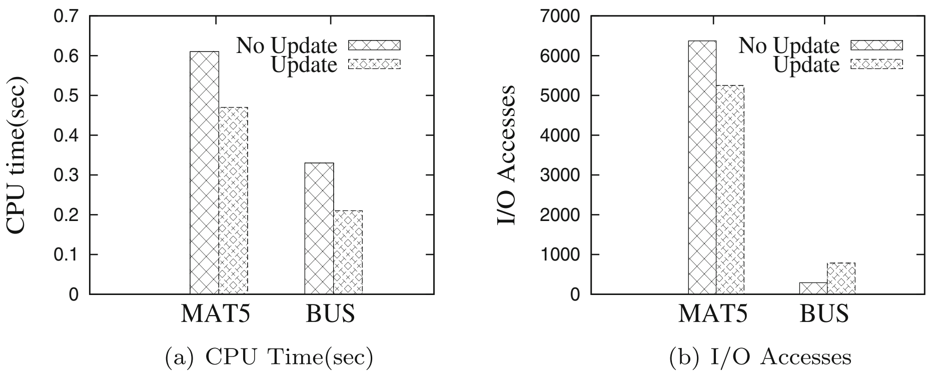


Fig. 38 Query performance

BAR The system is able to flexibly build the traditional trajectory index or the hybrid index, depending on whether standard trajectories or multi-attribute trajectories are processed. BAR is not tightly integrated into the spatio-temporal index and therefore can be combined with other traditional trajectory indexes, categorized into (i) R-tree based indexes, e.g., TB-tree [37], MV3R-Tree [45], and (ii) grid based indexes, e.g., SETI [6]. The well-established structures do not have to be modified, benefiting the system development. Figure 39 reports BAR built on top of TB-tree and Grid index, following a similar procedure to Section 3.3.1. We instantiate into the 3D R-tree due to the advantage of preserving the spatio-temporal proximity and the efficiency of answering nearest neighbor queries, as demonstrated in [21], and compare with other trajectory indexes as follows.

- TB-tree. The structure has the trajectory preservation property that only stores units of the same trajectory within a leaf node, resulting in a large spatial extent of the leaf nodes. The spatial proximity is not preserved because segments of different trajectories that lie spatially close will be stored in different nodes. We cannot effectively prune the search space by *min* and *max* distances, resulting in poor performance for nearest neighbor queries. The STR-tree [37] introduces a parameter to balance between spatial properties and trajectory preservation, but the main concern is to handle the spatial domain and treating the temporal as a secondary issue.
- SETI. The space is divided into disjoint cells, each of which contains trajectory segments that are completely within the cell and has a temporal index (an R*-tree) for objects' time intervals. The number of spatial partitions plays a crucial role in index design, but setting an optimal value is not trivial, e.g., trajectories may be uniformly or skewly distributed, making the performance unstable. The method focuses on the spatial proximity and has the limitation that the boundaries of the spatial dimension remain constant. The SEB-tree [41] is similar to SETI where the space is partitioned into zones, but the difference is that only the zone information is stored in the database without knowing the exact location.
- MV3R-tree. The index combines a multi-version R-tree (MVR-tree) and a small auxiliary 3D R-tree built on the leaf nodes of the MVR-tree. The former is to process timestamp queries and the latter is to process long interval queries. Short interval queries are processed by selecting the appropriate tree. We deal with interval queries and thus the structure is essentially a 3D R-tree. The MR-tree [58] builds separate R-trees for each timestamp and achieves good performance in the case of timestamp queries. However, the performance is not efficient for time window (interval) queries and there are many replicated node entries.

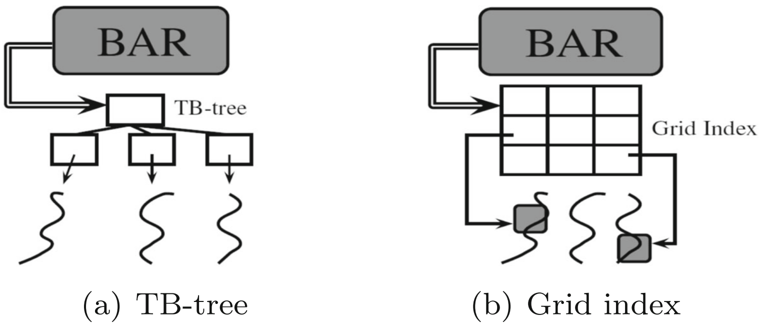


Fig. 39 Popularizing BAR

Queries The following queries can be answered following our framework: (i) spatio-temporal windows + attributes, e.g., "*Did any* RED BENZ *pass the restricted area during* [t_1, t_2]?"; (ii) continuous distance queries + attributes, e.g., "*Continuously report all TOYOTAs within 200 meters to the target?*"; and (iii) spatio-temporal similarity + attributes, e.g., "*Did any SILVER VW follow the target?*". We search BAR to find the subtrees in the spatio-temporal index fulfilling the attribute condition and then explore the spatio-temporal index. If attributes are not considered, the algorithm directly searches the spatio-temporal index without accessing BAR.

8 Related work

8.1 Querying trajectories

In the literature, tremendous efforts have been made on querying standard trajectories. Representative works include nearest neighbors [18, 19], similarity search [8, 42] and pattern discovery [27] [32]. In particular, continuous nearest neighbor queries are studied in [17, 21], but the result is different from C*k*NN_MAT.

In the era of big data, a large amount of trajectory data is collected and used to support applications such as personalized route recommendation [11]. The paper [31] studies retrieving distinct trajectories passing a user-specified spatio-temporal region over big trajectory data, and estimates the answers with a guaranteed error bound. Scalable algorithms for nearest neighbor joins are studied in [13]. The processed data, mainly retrieved from GPS, is standard trajectories, i.e., within the scope of location and time. Although GPS records contain a variety of attributes such as velocity, direction and acceleration, they are location-related and can be inferred from motion function. We enrich the data representation to support attributes independent of locations and allow users to find k nearest neighbors over time with certain attribute values, extending the query capability.

Emerging applications require extensive information about trajectories such as quality and semantics [65]. *Semantic trajectories* are studied to discover meaningful knowledge from locations [2, 59, 62]. A semantic enriched trajectory is typically defined to be a sequence of timestamped places, where each place is represented by a location with a semantic label. Interesting patterns can be properly defined and extracted. For example, a so-called *fine-grained sequential pattern* reports trajectories that satisfy spatial compactness, semantic consistency and temporal continuity simultaneously [61]. Consider actions that users can take at particular places such as sport, dining and entertaining. *Activity trajectories* are defined by associating geo-spatial points with activities. A similarity search returns k trajectories whose semantics contain the query and have the shortest minimum match distance [64]. Motivated by the fact that standard trajectories do not make much sense for humans, a *partition-and-summarization* approach is proposed to automatically generate texts to highlight the significant semantic behavior [43]. A good survey of semantic trajectories refers to [36].

Moving objects with transportation modes are investigated in [55, 57]. A trajectory over diverse geographical spaces includes timestamped locations and a sequence of transportation modes such as *Indoor*→ *Walk* →*Car*. Queries containing transportation modes can be answered, e.g., "*who arrived at the university by taxi*". A generic model is proposed to capture a wide range of meanings derived from a standard trajectory, called *symbolic trajectory* [23]. A systematic study of *annotated trajectory databases* is performed to represent a symbolic trajectory by a time-dependent label, which can be names of roads and speed profile, for example.

There are fundamental differences between those works and multi-attribute trajectories. First, we consider attributes that are location-independent and can not be discovered from standard trajectories. This differs from attaching location labels in semantic trajectories. Symbolic trajectories do not contain geo-locations, while multi-attribute trajectories do. Second, different queries are evaluated. We incorporate attributes into the evaluation for Boolean queries and search *k* nearest neighbors at each defined time. Previous queries deal with spatial closeness and attributes similarity instead of time-dependent distances and exact matches on attributes. Labels are sparsely defined in semantic trajectories because a few locations may contain semantics. As a result, ranking queries are primarily dealt with rather than continuous queries with attributes.

One more related work is heterogeneous *k*-nearest neighbor queries [44]. A moving object is represented by a location-independent attribute and a set of coordinates. By defining a function that combines the costs of distances and the location-independent attribute, the query returns objects having the *k*-th smallest value. Although the work considers the location-independent attribute, there are three major differences in comparison with ours. First, the data representation is limited in scope because each moving object is associated with only one attribute. We consider multiple attributes to have a generic solution. Second, they query objects based on a ranking function on distance and attribute, but we require exact matches on attribute, leading to different results. Third, their distance function is not time-dependent, while we deal with distances changing over time to support continuous queries. The query in [63] continuously reports *k* nearest spatial points with keywords to a moving object in a road network. The processed data is spatial points with keywords, but we deal with moving objects.

8.2 Indexing and manipulating trajectories

In the last decade, an impressive number of access methods have been proposed to optimize the processing of trajectory queries [6, 38, 39, 45, 48]. A survey of trajectory indexing and retrieval is given in [29]. These indexes only handle the proximity on spatial and temporal data, but do not achieve an optimal performance for C*k*NN_MAT. One one hand, we cannot use the index to select objects fulfilling the attribute condition because the spatio-temporal indexes do not manage attributes. On the other hand, the criteria of *minimum* and *maximum* distances cannot be used for pruning because objects with small distances may not contain query attribute values. Exemplified by Fig. 40, employing the pruning heuristic of *min* and

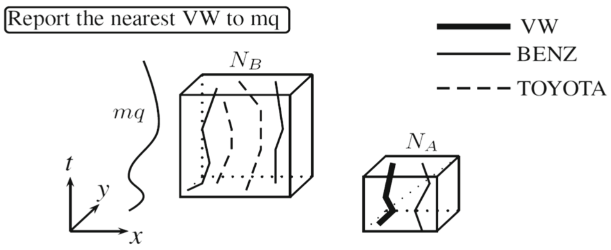


Fig. 40 Min and max distances

max distances, the node N_A is pruned because objects in N_A are further than N_B to mq. However, this produces false results because objects in N_B do not fulfill the attribute condition. Spatio-temporal index structures are not aware of attribute values and therefore cannot determine nodes containing query attribute values.

A grid index is established to organize activity trajectories in a hierarchical manner [64]. Activities are location-dependent and the index maintains the spatial and activity proximities. A similar structure is developed to incorporate both spatial and semantic information for approximate keyword search [62]. The grid index is a spatial index that maintains locations with activities rather than continuous movements. They address ranking queries, but we deal with continuous queries.

Several algorithms are proposed to minimize the total volume of trajectory approximations when given a user-specified number of splits [25]. *Rasetic et al.* provide an improved solution that splits trajectories into a number of sub-trajectories and indexes them to minimize the number of expected disk I/Os with respect to an average size of spatio-temporal range queries [39]. Calibrating trajectory data is studied in [42], the goal of which is to transform heterogeneous trajectories to one with unified sampling strategies. The method rewrites raw trajectories based on a reference set in order to remove the impact of the sampling heterogeneity. Trajectories after calibrating are not the same as the original form due to operations such as shifting, removing and inserting. The line of work on *trajectory simplification* [33] [34, 54] aims to approximate a trajectory by a simplified one with fewer points. The processing primitives are to retrieve less trajectories but in good quality. However, trajectories are modified and not the same as before. We merge small pieces of movements and approximate trajectories in a compact way without any information loss. The trajectory segmentation serves the purpose of partitioning each trajectory into sub-trajectories for a compact representation [62]. They evaluate spatial, time and semantic variances based on a local trajectory and aim to find an optimal method that uses the minimum number of trajectory segments such that the maximum decrease of feature value is less than a threshold. The paper [12] optimizes the maintenance of continuous queries over standard trajectories.

Moving objects databases, as the representative of update-intensive applications, need to handle frequent updates. A number of approaches have been proposed to improve the efficiency of updating spatio-temporal indexes [5, 26, 40]. To support updating multi-attribute trajectories, not only locations and time but also attribute values are addressed. Meanwhile, low latency should be achieved to allow instantaneous queries.

8.3 Spatial keyword queries

Recently, spatial keyword queries (SKQ for short) have been extensively studied in the literature [7, 30, 52]. The task is to support queries that take a geo-location and a set of text descriptions called keywords as arguments and return objects that are close to the query location and contain the keywords called Boolean kNN query [14], or objects with the highest ranking scores measured by a combination of distances to the query location and the text relevance to the keywords called Top-k NN query [10]. To efficiently answer the query, a spatial index such as 2D R-tree and a text index structure are combined. For example, the IR-tree [10] arguments each node of the R-tree with a pointer pointing to an inverted file that contains a summary of the text content of the objects in the corresponding subtree. During the query procedure, one uses the combined structure to estimate both the spatial distance and the text relevancy and prune the objects that cannot contribute to the result.

Compared to SKQ, multi-attribute trajectories are in principle the combination of standard trajectories and attributes. Both problems extend the traditional spatial and moving objects to enrich the data representation. However, there are some fundamental differences. In the aspect of data representation, SKQ focuses on static geo-locations and location-dependent text descriptions. We cope with moving objects and location-independent attributes, leading to different queries. SKQ considers a static query location, while our query is a moving object and returns the nearest object to the query at each query time, complicating the evaluation. Furthermore, text descriptions and attributes will make different tasks when designing the index structure. In SKQ, the index groups close objects in terms of spatial distances and location-related text relevances. Attributes do not depend on particular locations but are associated with objects. It is possible to attach attributes to timestamped locations, but each piece of trajectories will have all attributes along with the trajectory, resulting in an extremely large amount of redundant data. In fact, the key issue of boosting the index is to know which objects contain particular attribute values and where they are located in the spatio-temporal index. Therefore, a different criterion is used to build the index. Last, there is no update in SQK, but we accommodate frequent updates.

9 Conclusions

We enrich the trajectory representation to form multi-attribute trajectories and propose a new query. A hybrid index structure is designed and updating the index is also supported. We make a systematic design such that the attribute structure can be combined with a range of standard trajectory indexes. Efficient query algorithms are developed. An extensive experimental evaluation is conducted on large datasets to demonstrate the performance advantage of our approach over baseline methods. The future work is to consider similarity queries on multi-attribute trajectories.

Acknowledgments We sincerely thank Fabio Valdés, Thomas Behr and Sara Betkas for their helpful comments to improve the preliminary version. This work is supported by National Key Research and Development Plan of China (2018YFB1003902) and the Fundamental Research Funds for the Central Universities (NO. NS2017073).

Appendix

Unique attribute values by composite numbers Given a point (x, y), its Z-order value is denoted by z-val(x, y) and the binary representation is $z[2\cdot m] : z[i] = x[i], z[i+1] = y[i]$, $i \in [0, m]$, $x[m]$, $y[m]$ are arrays of bits for binary representations x and y, respectively, and m is the number of bits to represent the coordinates.

Lemma 6 *Let $a_1 \in dom(A_{d_1})$ and $a_2 \in dom(A_{d_2})$ be attribute values from two different domains, respectively. Then, we have z-val$(d_1, a_1) \neq$ z-val(d_2, a_2).*

Proof Let $z_1[2 \cdot m]$ and $z_2[2 \cdot m]$ be binary representataions for z-val(d_1, a_1) and z-val(d_2, a_2), respectively. Because of $d_1 \neq d_2$, then arrays $x_1[m]$ and $x_2[m]$ are not equal. After the interleaving, there exists an even bit $i \in [0, 2 \cdot m - 1]$ such that $z_1[i] \neq z_2[i]$. As a result, we have $z_1[2 \cdot m] \neq z_2[2 \cdot m]$. The condition holds regardless of a_1 and a_2. □

References

1. (2016). http://factory.datatang.com/en/
2. Alvares LO, Bogorny V, Kuijpers B et al (2007) Towards semantic trajectory knowledge discovery. Data Mining and Knowledge Discovery
3. Bentley JL, Ottmann T (1979) Algorithms for reporting and counting geometric intersections. IEEE Trans Computers 28(9):643–647
4. Bercken J, Seeger B, Widmayer P (1997) A generic approach to bulk loading multidimensional index structures. In: VLDB, pp 406–415
5. Biveinis L, Saltenis S, Jensen C (2007) Main-memory operation buffering for efficient r-tree update. In: VLDB, pp 591–602
6. Chakka VP, Everspaugh A, Patel JM (2003) Indexing large trajectory data sets with seti. In: CIDR
7. Chen L, Cong G, Jensen C, Wu D (2013) Spatial keyword query processing: an experimental evaluation. PVLDB 6(3):217–228
8. Chen L, Özsu MT, Oria V (2005) Robust and fast similarity search for moving object trajectories. In: SIGMOD, pp 491–502
9. Chen Z, Shen H, Zhou X, Zheng Y, Xie X (2010) Searching trajectories by locations-an efficiency study. In: SIGMOD, pp 255–266
10. Cong G, Jensen C, Wu D (2009) Efficient retrieval of the top-k most relevant spatial web objects. PVLDB 2(1):337–348
11. Dai J, Yang B, Guo C, Ding Z (2015) Personalized route recommendation using big trajectory data. In: ICDE, pp 543–554
12. Ding H, Trajcevski G, Scheuermann P (2008) Efficient maintenance of continuous queries for trajectories. GeoInformatica 12(3):255–288
13. Fang Y, Cheng R, Tang W, Maniu S, Yang XS (2016) Scalable algorithms for nearest-neighbor joins on big trajectory data. IEEE Trans Knowl Data Eng 28(3):785–800
14. Felipe I, Hristidis V, Rishe N (2008) Keyword search on spatial databases. In: ICDE, pp 656–665
15. Forlizzi L, Güting RH, Nardelli E, Schneider M (2000) A data model and data structures for moving objects databases. In: SIGMOD, pp 319–330
16. Frentzos E, Gratsias K, Pelekis N, Theodoridis Y (2005) Nearest neighbor search on moving object trajectories. In: SSTD, pp 328–345
17. Frentzos E, Gratsias K, Pelekis N, Theodoridis Y (2007) Algorithms for nearest neighbor search on moving object trajectories. GeoInformatica 11(2):159–193
18. Gao Y, Zheng B, Chen G, Li Q (2010) Algorithms for constrained k-nearest neighbor queries over moving object trajectories. GeoInformatica 14(2):241–276
19. Gao Y, Zheng B, Chen G, Li Q, Guo X (2011) Continuous visible nearest neighbor query processing in spatial databases. VLDB J 20(3):371–396
20. Güting RH, Behr T, Düntgen C (2010) SECONDO: a platform for moving objects database research and for publishing and integrating research implementations. IEEE Data Eng Bull 33(2):56–63
21. Güting RH, Behr T, Xu J (2010) Efficient k-nearest neighbor search on moving object trajectories. VLDB J 19(5):687–714
22. Güting RH, Böhlen M, Erwig M, Jensen C, Lorentzos N, Schneider M, Vazirgiannis M (2000) A foundation for representing and querying moving objects. ACM TODS 25(1):1–42
23. Güting RH, Valdës F, Damiani M (2015) Symbolic trajectories. ACM Trans Spatial Algo Syst, 1(2):Article 7
24. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: SIGMOD, pp 47–57
25. Hadjieleftheriou M, Kollios G, Tsotras VJ, Gunopulos D (2002) Efficient indexing of spatiotemporal objects. In: EDBT, pp 251–268
26. Jensen C, Lin D, Ooi BC (2017) Maximum update interval in moving objects databases. In: Encyclopedia of GIS, p 1205
27. Jeung H, Yiu M, Zhou X, Jensen C, Shen H (2008) Discovery of convoys in trajectory databases. PVLDB 1(1):1068–1080
28. Lange R, Dürr F, Rothermel K (2011) Efficient real-time trajectory tracking. VLDB J 20(5):671–694
29. Dinh L, Aref WG, Mokbel MF (2010) Spatio-temporal access methods: Part 2 (2003-2010). IEEE Data Eng Bull 33(2):46–55
30. Lee T, Park J, Lee S, Hwang S, Elnikety S, He Y (2015) Processing and optimizing main memory spatial-keyword queries. PVLDB 9(3):132–143
31. Li Y, Chow C, Deng K, Yuan M, Zeng J, Zhang J, Yang Q, Zhang Z (2015) Sampling big trajectory data. In: CIKM, pp 941–950

32. Li Z, Ding B, Han J, Kays R (2010) Swarm: mining relaxed temporal moving object clusters. PVLDB 3(1):723–734
33. Long C, Wong RC, Jagadish HV (2013) Direction-preserving trajectory simplification. PVLDB 6(10):949–960
34. Long C, Wong RC, Jagadish HV (2014) Trajectory simplification: on minimizing the direction-based error. PVLDB 8(1):49–60
35. Mauroux PC, Wu E, Madden S (2010) Trajstore: an adaptive storage system for very large trajectory data sets. In: ICDE, pp 109–120
36. Parent C, Spaccapietra S, Renso C et al (2013) Semantic trajectories modeling and analysis. ACM Comput Surv 45(4):42
37. Pfoser D, Jensen C (2000) Novel approaches in query processing for moving object trajectories. In: VLDB, pp 395–406
38. Popa IS, Zeitouni K, Oria V, Barth D, Vial S (2011) Indexing in-network trajectory flows. VLDB J 20(5):643–669
39. Rasetic S, Sander J, Elding J, Nascimento MA (2005) A trajectory splitting model for efficient spatio-temporal indexing. In: VLDB, pp 934–945
40. Sidlauskas D, Saltenis S, Jensen C (2014) Processing of extreme moving-object update and query workloads in main memory. VLDB J 23(5):817–841
41. Song Z, Roussopoulos N (2003) Seb-tree: An approach to index continuously moving objects. In: MDM, pp 340–344
42. Su H, Zheng K, Wang H, Huang J, Zhou X (2013) Calibrating trajectory data for similarity-based analysis. In: SIGMOD, pp 833–844
43. Su H, Zheng K, Zeng K, Huang J, Sadiq SW, Yuan N, Zhou X (2015) Making sense of trajectory data A partition-and-summarization approach. In: ICDE, pp 963–974
44. Su Y, Wu Y, Chen ALP (2007) Monitoring heterogeneous nearest neighbors for moving objects considering location-independent attributes. In: DASFAA, pp 300–312
45. Tao Y, Papadias D (2001) Mv3r-tree: a spatio-temporal access method for timestamp and interval queries. In: VLDB, pp 431–440
46. Tao Y, Papadias D, Shen Q (2002) Continuous nearest neighbor search. In: VLDB, pp 287–298
47. Tong Y, Chen L, Zhou Z et al (2018) Slade: a smart large-scale task decomposer in crowdsourcing. IEEE Transactions on Knowledge and Data Engineering to appear
48. Tong Y, Chen Y, Zhou Z et al (2017) The simpler the better: a unified approach to predicting original taxi demands based on large-scale online platforms. In: ACM SIGKDD, pp 1653–1662
49. Tong Y, She J, Ding B, Wang L, Chen L (2016) Online mobile micro-task allocation in spatial crowdsourcing. In: ICDE, pp 49–60
50. Tzoumas K, Yiu ML, Jensen C (2009) Workload-aware indexing of continuously moving objects. PVLDB 2(1):1186–1197
51. Wang H, Zimmermann R (2011) Processing of continuous location-based range queries on moving objects in road networks. IEEE Trans Knowl Data Eng 23(7):1065–1078
52. Wang X, Zhang Y, Zhang W, Lin X, Huang Z (2016) SKYPE: Top-k spatial-keyword publish/subscribe over sliding window. PVLDB 9(7):588–599
53. Wu D, Yiu ML, Cong G, Jensen C (2012) Joint top-k spatial keyword query processing. IEEE Trans Knowl Data Eng 24(10):1889–1903
54. Lin HZTWX, Ma S, Huai J (2017) One-pass error bounded trajectory simplification. PVLDB 10(7):841–852
55. Xu J, Güting R, Zheng Y (2015) The TM-RTree: an index on generic moving objects for range queries. GeoInformatica 19(3):487–524
56. Xu J, Güting RH (2012) MwgenG: a mini world generator. In: MDM, pp 258–267
57. Xu J, Güting RH (2013) A generic data model for moving objects. GeoInformatica 17(1):125–172
58. Xu X, Han J, Lu W (1990) Rt-tree: an improved r-tree indexing structure for temporal spatial databases. In: SDH, pp 1040–1049
59. Yan Z, Chakraborty D, Parent C, Spaccapietra S, Aberer K (2011) Semitri: a framework for semantic annotation of heterogeneous trajectories. In: EDBT, pp 259–270
60. Yao B, Xiao X, Li F, Wu Y (2014) Dynamic monitoring of optimal locations in road network databases. VLDB J 23(5):697–720
61. Zhang C, Han J, Shou L, Lu J, Porta TFL (2014) Splitter: mining fine-grained sequential patterns in semantic trajectories. PVLDB 7(9):769–780
62. Zheng B, Yuan N, Zheng K, Xie X, Sadiq SW, Zhou X (2015) Approximate keyword search in semantic trajectory database. In: ICDE, pp 975–986

63. Zheng B, Zheng K, Xiao X, Su H, Yin H, Zhou X, Li G (2016) Keyword-aware continuous knn query on road networks. In: IEEE ICDE, pp 871–882
64. Zheng K, Shang S, Yuan N, Yang Y (2013) Towards efficient search for activity trajectories. In: ICDE, pp 230–241
65. Zheng K, Su H (2015) Go beyond raw trajectory data: quality and semantics. IEEE Data Eng Bull 38(2):27–34
66. Zheng K, Zheng Y, Yuan N, Shang S (2013) On discovery of gathering patterns from trajectories. In: ICDE, pp 242–253

Jianqiu Xu got his bachelor and master degree from Nanjing University of Aeronautics and Astronautics in 2005 and 2008, respectively. Then, he studied the Ph.D supervised by Prof. Dr. Ralf Hartmut Güting between 2008.9 and 2012.10 from FernUniversität in Hagen, Germany, focusing on moving objects databases and spatial databases. In 2013.1, he joined Nanjing University of Aeronautics and Astronautics in China as an assistant professor.

Ralf Hartmut Güting has been a full professor in Computer Science at the University of Hagen, Germany, since 1989. He received his Diploma and Dr. rer. nat. degrees from the University of Dortmund in 1980 and 1983, respectively, and became a professor at that university in 1987. From 1981 until 1984 his main research area was Computational Geometry. After a one-year stay at the IBM Almaden Research Center in 1985, extensible and spatial database systems became his major research interests; more recently, also spatio-temporal or moving objects databases. He has been an associate editor of the ACM Transactions on Database Systems and an editor of the VLDB Journal and is on the Editorial Board of GeoInformatica. He has published two German text books on data structures and algorithms and on compilers, respectively, and an English text book on moving objects databases, as well as around eighty journal and conference articles. His group has built prototypes of extensible and spatio-temporal database systems, the Gral system and the SECONDO system.

Yunjun Gao received the PhD degree in computer science from Zhejiang University, China, in 2008. He is currently an associate professor in the College of Computer Science, Zhejiang University, China. His research interests include spatial and spatio-temporal databases, metric and incomplete/uncertain data management, and spatio-textual data processing. He is a member of the ACM and the IEEE, and a senior member of the CCF.