Geoinformatica (2017) 21:429–458
DOI 10.1007/s10707-016-0286-6

Index-supported pattern matching on tuples of time-dependent values

Fabio Valdés[1] · **Ralf Hartmut Güting**[1]

Received: 15 April 2016 / Revised: 22 September 2016 / Accepted: 8 December 2016/
Published online: 3 January 2017

Abstract Lately, the amount of mobility data recorded by GPS-enabled (and other) devices has increased drastically, entailing the necessity of efficient processing and analysis methods. In many cases, not only the geographic position, but also additional time-dependent information are traced and/or generated, according to the purpose of the evaluation. For example, in the field of animal behavior research, besides the position of the monitored animal, biologists are interested in further data like the altitude or the temperature at every measuring point. Other application domains comprise the names of streets, places of interest, or transportation modes that can be recorded along with the geographic position of a person. In this paper, we present in detail a framework for analyzing datasets with arbitrarily many time-dependent attributes. This can be considered as a major extension of our previous work, a comprehensive framework for pattern matching on symbolic trajectories with index support. For an efficient processing of different data types, a variable number of indexes of four different types that correspond to the data types of the attributes are applied. We demonstrate the expressiveness and efficiency of our approach by querying a real dataset representing taxi trips in Rome and, particularly, with a broad series of experiments using trajectories generated by BerlinMOD combined with geological raster data.

Keywords Pattern matching · Tuples of time-dependent values · Indexing · Finite automaton

✉ Fabio Valdés
fabio.valdes@fernuni-hagen.de

Ralf Hartmut Güting
rhg@fernuni-hagen.de

[1] Database Systems for New Applications, Fernuniversität in Hagen, 58084, Hagen, Germany

1 Introduction

Since position recording devices like smart phones, tablets, automotive navigation systems, or other GPS sensors are applied abundantly for economic, scientific, private, or other purposes, an overwhelming amount of movement data is generated every day. As a consequence, researchers strive to enhance methods for storing, administrating, and querying such data. The sequence of timestamped geographic positions obtained from the device represents the movement of an entity, e.g., the trajectory of a person, a vehicle, or an animal, during a certain period of time. More abstractly, the recorded data can be considered as a continuous function from time into two-dimensional space, denoted as moving point [14].

A symbolic representation of the movement, in most cases shorter than the raw trajectory and often derived from it, enables more convenient and more efficient querying [16]. However, for many applications, more time-dependent values than the mere movement data are helpful or even necessary. For example, the observation of animals is definitely more effective if not only their position but also additional data like the temperature or the altitude are available. Besides research fields like animal behavior analysis, possible application domains include economics (e.g., logistical optimization, customer behavior analysis, targeted advertising) as well as urban planning, healthcare, private use, and criminal investigation. Each of the available data types is well-defined as an abstract function and has a discrete representation that is available in the DBMS SECONDO.

Usually, a symbolic representation of a movement is given as a sequence of labels, for example, street names, transportation modes, names of places of interest, districts, or cells inside a cellular network, along with some temporal information and in chronological order. Therefore, it seems natural to develop an expressive pattern language for querying such trajectories that allows for regular expressions and employs a nondeterministic finite automaton for the matching algorithm. However, if the user desires to define more complex patterns including variables and conditions, more sophisticated techniques with possibly high computation cost are required. In order to keep the pattern matching efficient, the use of an index structure for all the labels in the trajectory collection is mandatory, so the corresponding matching algorithm does not have to scan all the trajectories. Starting from this stage of development, we decided to extend our work in order to meet the requirements described in the previous paragraph.

In this paper, we introduce a framework for analyzing sets of tuples having an arbitrary number of time-dependent attributes of different data types. As the major contribution of this paper, we developed and implemented a suitable pattern matching algorithm that filters a set of tuples according to a user-specified pattern. The pattern language established in [16] and revised in [36] has been further extended, so that the user can query all the time-dependent attributes of the tuple collection at the same time. For an efficient processing, we apply a flexible combination of well-known index structures. The combined index structure comprises a variable number of heterogeneous indexes, corresponding to the number and the data types of the time-dependent attributes, and is constructed efficiently from the tuple collection.

In the first phase of the pattern matching algorithm, we prune the tuples for which the multi-index does not yield any results related to certain components of the pattern. The exact pattern matching is then performed on the reduced dataset. While executing the NFA transition function generated from the pattern, information about the matching as well as multi-index retrievals are held and updated inside specialized efficient data structures. Note that without index support, every time-dependent value inside each tuple in general would have to be scanned completely for a matching decision. The index-supported approach

detailed in this paper reduces the computation cost by one or even two orders of magnitude, for which one reason is the lower influence of the size of the time-dependent values on the runtime. We apply the presented data structures and algorithms by means of a continuous example. For a comprehensive experimental evaluation, we created collections of geographic trajectories having different properties with the help of the moving objects databases benchmark BerlinMOD [10]. These trajectories were combined with geological elevation data of Berlin from the U.S. Geological Survey.

In the subsequent list, we summarize the contributions of this paper:

- We provide a framework for analyzing sets of tuples that have any number of time-dependent values.
- The corresponding pattern matching algorithm is fully implemented and available as a module of the DBMS SECONDO.
- We detail an extension of the existing pattern language that was only suitable for symbolic trajectories.
- We apply a flexible multi-index for all time-dependent attributes.
- We introduce sophisticated data structures for efficient processing.
- An application example based on a real dataset is presented.
- With the help of synthesized data, we compare our approach to a baseline algorithm without index support.

The remainder of the paper is organized as follows: The subsequent section is dedicated to related work. After Section 3 introduces time-dependent values as well as the corresponding pattern language and gives an insight into the DBMS SECONDO, in Section 4 we discuss the multi-index for sets of tuples of time-dependent values and further auxiliary data structures. Section 5 presents the algorithms that are applied during the preprocessing and the pattern matching. An application example with a real dataset is detailed in Section 6. We provide the experimental evaluation in Section 7, before Section 8 concludes the paper.

2 Related work

In the last 15 years, the research field of moving objects databases has been highly active [15, 45]. A conceptual trajectory model, based on the key concepts of stop and move, is first defined in [33], followed by generalized variants [2, 28, 42] of this model. However, since they focus on the conceptual level, issues of data management remain unsolved. A comprehensive framework for the generalized representation of movement in a symbolic space is introduced in [16]. It is fully integrated into the data model of [14], available for moving objects database systems such as SECONDO [1, 13] or Hermes [29]. The framework is also embedded into the SECONDO implementation of the model of [14]. It includes four data types for different kinds of symbolic trajectories and an expressive and fully implemented pattern matching language. Demonstrations applying a set of private trajectories of a person and the Microsoft GeoLife dataset [21] are provided in [35] and [7], respectively. The most recent demonstration [37] focuses on the analysis of animal movement with the help of the approach detailed in this paper. Another publication [8] illustrates the application potential of symbolic trajectories.

A symbolic trajectory corresponds to a sequence of strings (or sets of strings), so the use of regular expressions for a pattern language and thus finite automata [19, 24] for pattern matching algorithms on symbolic trajectories seems logical. In [22, 23], a pattern language for trajectories defined in a discrete symbolic space and a pattern matching algorithm based on

a nondeterministic finite automaton (NFA) are presented. An object's trajectory is defined as a sequence of symbols denoting the successive zones the considered object visited. The drawback of the proposed language is the lack of precise temporal specifications or conditions on variables. The authors of [39, 40] provide an expressive pattern language for geometric trajectories that allows variables and conditions. However, their approach is limited to symbolic trajectories containing names of areas inside a partitioned space. In [18], a precursor to their work, more general geometries are considered, not just partitions of the plane into regions. On the other hand, this approach focuses on range and nearest-neighbor queries, does not introduce any symbolic representation, and does not allow for variables or regular expressions. The model of [44] does not include time at all and otherwise is a sequence of timestamped locations annotated with a semantic label, where timestamps are instants of time, not time intervals. The detection of frequent sequential patterns (i.e., temporally bounded transitions from one or more places to another group of places) from a set of semantic trajectories is supported by [43], where similar places (regarding spatial, semantic, and temporal aspects) are grouped together.

The application of index structures is appropriate for realizing efficient search queries and pattern matching algorithms on trajectories. Hence, index structures for spatial trajectories are explored in several publications, e.g., the 3D R-tree [38], the TB-tree [30], or the TMN-tree [5]. The authors of [26] give an overview of indexes for spatio-temporal data of several categories. An index structure for discovering similar multidimensional trajectories is detailed in [41]. In [36], a twofold index structure for symbolic trajectories and a new matching algorithm are presented, extending the framework established in [16] and allowing efficient pattern matching queries whose runtime does not depend on the trajectories' length anymore. The authors of [20] introduce an index for spatial trajectories with additional labels. Their approach is efficient and includes a suitable query language which is rather limited as it does not allow regular expressions, conditions, or the specification of time intervals.

To the best of our knowledge, by now there is no publication on a pattern matching algorithm for collections of tuples of arbitrarily many time-dependent attributes of different data types supported by a multi-index which is realized as a flexible combination of classic index structures. All existing index structures for spatial or symbolic trajectories are limited and cannot process several attributes of different data types.

3 Preliminaries

In this section, we give an introduction to symbolic trajectories and, more generally, to time-dependent data types in abstract and discrete representation, reviewing some of the results of [16]. After an overview of the DBMS SECONDO, we mention basic notations, and we describe the extensions that the new pattern language offers. Finally, a pattern is defined which we use as a continuous example for illustrating the employed data structures and algorithms.

3.1 Symbolic trajectories and pattern matching

First, we provide a short example for a symbolic trajectory, describing the sequence of streets that a person has passed during her/his trip through Beverly Hills, California:

[2015-12-16-19:09:09	2015-12-16-19:11:58)	Wilshire Blvd
[2015-12-16-19:11:58	2015-12-16-20:01:22)	Santa Monica Blvd
[2015-12-16-20:01:27	2015-12-16-20:07:41)	Rodeo Dr
[2015-12-16-20:07:41	2015-12-16-22:49:09)	Sunset Blvd

It consists of four so-called units, each of which is a pair of a time interval and a label. The brackets and parentheses indicate whether or not a time interval is leftclosed and/or rightclosed. Note that the time intervals have to be disjoint but not necessarily continuous. In the database system SECONDO, its data type is *moving(label)*, or *mlabel*, for short. An object of the data type *mlabel* represents a time-dependent *label*, i.e., a character string that changes its values over time.

Database systems with support for this or similar data types offer many possibilities for querying such a symbolic trajectory, e.g., in SECONDO operations like **passes** or **atinstant** can be applied to find out whether or not certain predicates are fulfilled. However, for more complex requests, a corresponding query is often hardly expressible and/or inefficiently executed. Therefore, we developed a highly expressive but rather simple pattern language and an algorithm that determines whether a symbolic trajectory M is matched by a pattern p. Such a pattern consists of pattern atoms that can match either a sequence of units of M or precisely one unit u, if the given specifications in that atom a match u. The latter "match" means – in the basic version – that the time interval $t(u)$ of u is completely covered by the temporal period $t(a)$ specified in a, i.e., $t(u) \subset t(a)$, and that every label in u also occurs in a, that is, $l(u) \subset l(a)$. Other set relations than 'subset' can be chosen, the alternatives are 'disjoint', 'superset', 'equal', and 'intersect'. The order of the atoms in p has to correspond to the order of matched units in M. For example, the pattern

```
X (wednesday "Wilshire Blvd") Y [* (_ "Sunset Blvd")]
// get_duration(X.time) < get_duration(Y.time)
```

matches the abovementioned trajectory, since the latter starts at Wilshire Blvd on a Wednesday, ends at Sunset Blvd without further temporal constraint, and has arbitrarily many units between them (the $*$ represents a wildcard). The condition after the double slash compares the temporal durations of the unit bound to X and the unit sequence bound to Y, respectively. It is fulfilled, since the person spent less time on Wilshere Blvd than during the remaining trajectory. The square brackets indicate that Y is associated to all atoms between them. The atom in parentheses can match exactly one trajectory unit. Such an atom consists of two or more (the precise number depends on the applied dataset) so-called atom values. Any number of conditions may be specified (separated by commas), and every operator of the underlying database system can be used; note that **get_duration** is not a part of the pattern language, but a SECONDO operator. This pattern language for querying a set of symbolic trajectories is detailed in [16] and slightly enhanced in [36].

3.2 Representing time-dependent data types

In this section, we give an introduction to abstract and discrete representations of time-dependent data types, reproducing some of the statements of [16] that are based on a comprehensive framework for representing and querying moving objects in databases [11, 12, 14].

The general idea is to provide a collection of abstract data types to describe moving objects and the operations applicable to them. For example, *movingpoint* (or *mpoint*, for short) is a data type to represent a time-dependent location in the Euclidean plane, *line* is a spatial data type describing a continuous curve in the plane, and *mreal* is a type to represent time-dependent real values. The operation **trajectory** maps a moving point to a *line* value and the **distance** operation, applied to two *mpoint* values, returns their time-dependent distance as an *mreal*.

These and many more data types and operations are embedded into the data model of a DBMS as follows. The data types can be used as attribute types. Hence, we may have a relation describing car trips with the schema

Vehicles (Id: *int*, Trip: *mpoint*).

The operations can be used in queries. For example, one can find pairs of vehicles that have been closer to each other than 100 meters by the query

```
SELECT v1.Id, v2.Id,
FROM Vehicles as v1, Vehicles as v2
WHERE minimum(distance(v1.Trip, v2.Trip)) < 0.1
```

which uses a further operation **minimum** that maps an *mreal* to a *real*.

Formally, a system of types and operations is a (many-sorted) algebra. It consists of a signature which provides sorts and operations, defining for each operation the argument sorts and the result sort. A signature defines a set of terms. To define the semantics, one needs to assign carrier sets to the sorts and functions to the operations that are mappings on the respective carrier sets. The signature together with carrier sets and functions defines the algebra.

In the framework discussed, data types are built from some basic types and type constructors. The type system is itself described by a signature. In this signature, the sorts are so-called kinds and the operations are type constructors. The terms of the signature are exactly the available types of the type system. For example, consider a signature

int, *real*, *bool*:		→ BASE
array:	BASE	→ ARRAY

It has the kinds BASE and ARRAY and the type constructors *int*, *real*, *bool*, and *array*. The types defined are the terms of the signature, namely, *int*, *real*, *bool*, *array*(*int*), *array*(*real*), and *array*(*bool*). Note that the basic types are just type constructors without arguments.

The type system defined in [14] for moving objects is shown in Table 1.

Table 1 Type system defined in [14]

Type Constructor	Signature	
int, *real*, *string*, *bool*		→ BASE
point, *points*, *line*, *region*		→ SPATIAL
instant		→ TIME
moving, *intime*	BASE ∪ SPATIAL	→ TEMPORAL
range	BASE ∪ TIME	→ RANGE

We first explain the type system informally. It has some basic standard types and some spatial data types. The type *instant* represents the continuous domain of time. The type constructor *moving* provides for a given static type a corresponding time dependent type. The *intime* constructor yields for a static type α a type whose values are pairs of an *instant* and a value of type α. The *range* constructor provides for a given type another type whose values are finite sets of disjoint intervals over the domain of α.

To provide formally the semantics of the data types, one needs to define their domains, or carrier sets. An important distinction introduced in [11, 12, 14] is that between an abstract model and a discrete model. In an abstract model, the domain may be defined in terms of infinite sets. Such a model is conceptually simple, but it is not directly implementable. In contrast, in a discrete model, the possible values of a data type must be defined in terms of finite representations. These can be mapped to data structures in the implementation.

For example, a *region* data type can be defined in an abstract model as a regular closed subset of the Euclidean plane, whereas in a discrete model it would be defined as a collection of disjoint polygons each of which may have polygonal holes.

The reference [14] defines an abstract model of data types and operations for moving objects whereas [12] provides a corresponding discrete model.

Some notations used in defining semantics of types are A_α and D_α to denote the carrier set of the type α in the abstract and discrete model, respectively. When a carrier set A_α contains an undefined value $\perp$, then the notation $\bar{A}_\alpha$ refers to the carrier set without the undefined value, i.e., $\bar{A}_\alpha = A_\alpha \setminus \{\perp\}$. With these notations, the carrier set of the *moving* type constructor is defined as follows.[1]

Definition 1 Given a data type α to which type constructor *moving* is applicable, the carrier set of the type *moving*(α) is

$$A_{\underline{moving}(\alpha)} := \{f \mid f : A_{\underline{instant}} \to \bar{A}_\alpha \text{ is a partial function}\}$$

Note that the abstract model disregards completely the issue of how such functions can be represented. A function $f : A_{\underline{instant}} \to A_\alpha$ is simply an infinite set of pairs from $A_{\underline{instant}} \times A_\alpha$.

The discrete model of [12] provides finite representations for all the types of the abstract model. For types *moving*(α) the so-called sliced representation is introduced. That means, to represent a function of time, the time domain is cut into disjoint time intervals (slices) such that within each slice the development can be represented by some simple function of time. "Simple" actually means finitely representable. In other words, the function for a slice can be described by a few parameters rather than an infinite set of pairs. Figure 1 illustrates the sliced representation for a *moving*(*real*) and a *moving*(*point*).

The representation of a single slice, consisting of the time interval and the function description, is called a unit.

For the given data types, a comprehensive set of operations is defined. Most of them are generic and applicable to many of the available data types. Two examples are

deftime:	*moving*(α)	$\to$ *periods*
atinstant:	*moving*(α) $\times$ *instant*	$\to$ *intime*(α)

[1] The definition in [14] has an additional condition requesting that such a function has only a finite number of continuous components, omitted here.

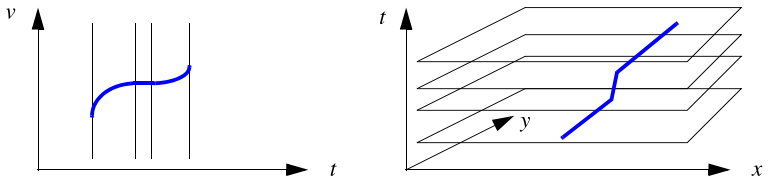

Fig. 1 Sliced representations for *moving*(*real*) and *moving*(*point*)

Here the type *periods* is just an abbreviation of *range*(*instant*). Hence the operation **deftime** returns the set of time intervals during which a moving object is defined. These two example operations are generic because they range over the types generated by the type constructor *moving*.

Finally, lifting is introduced as a mechanism to make static and related time-dependent operations consistent. Lifting means that for a given static operation each of the arguments may become time-dependent (i.e., replacing in the signature type α by *moving*(α)) which makes the result time-dependent as well. Furthermore, the semantics of the lifted operation is derived using the semantics of the static operation for every instant of time. Hence by lifting we also have an operation **inside**: *moving*(*point*) $\times$ *region* $\rightarrow$ *moving*(*bool*).

See [14] for further details and the complete definition of types and operations.

3.3 The DBMS SECONDO

This section provides a brief introduction to the DBMS SECONDO, reviewing some of the results of [13].

SECONDO is a prototype DBMS developed at University of Hagen since about 1995. It runs on Linux and MacOS X platforms and is freely available open source software [32]. The main design goals were a clean extensible architecture and support for spatial and spatio-temporal applications.

The architecture of SECONDO is depicted in Fig. 2. It consists of three major components: the kernel, the optimizer, and the GUI. The kernel does not implement a fixed data model but is open for the implementation of a wide variety of DBMS data models. The kernel is extensible by algebra modules. To be precise, the entire implementation of a particular data model is done within algebra modules. Such a module encapsulates everything

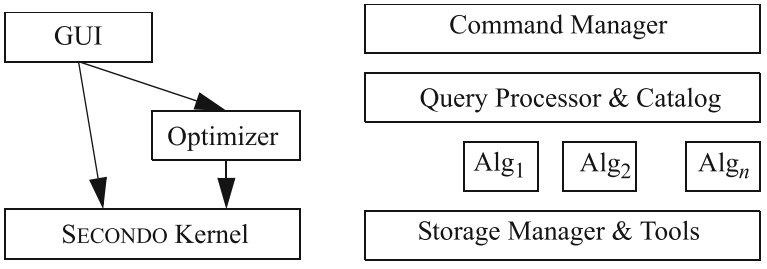

Fig. 2 SECONDO components (*left*), architecture of kernel system (*right*) [13]

needed to implement a DBMS data model, hence there are algebras in SECONDO for basic data types, for relations and tuples including operations such as hashjoin, for B-trees and R-trees with their access operations, for spatial and spatio-temporal data types, and many more. There are also algebras beyond the scope of a relational model such as for nested relations, networks, or parallel processing.

The SECONDO kernel is an engine to evaluate terms over the existing objects and operations. For example, it evaluates the expressions

```
query Trains feed filter[day_of(inst(initial(.Trip))) = 9]
                 filter[length(trajectory(.Trip)) > 2000.0]
count
```

where Trains is a relation containing trajectories represented in an attribute Trip of the type *mpoint*. The syntax for operations can be freely chosen, often it is convenient to use postfix notation for query processing operations. For example, the first argument to the **filter** operation is `Trains feed`. Stream processing is built into the engine. The commands and queries processed directly by the kernel are called the executable language. The kernel is written in C++ and uses BerkeleyDB as the underlying storage manager.

The optimizer is not as independent from the data model as the kernel. It assumes an object-relational model and supports an SQL-like language. It maps SQL to the executable language shown above. The optimizer is extensible by registering types and operations of the executable level, by translation rules and cost functions. It allows for extension by new index types, providing concepts to distinguish between logical and physical indexes (a physical index is a particular index structure available in the kernel, a logical index is a strategy to use it which may be complex). The optimizer determines predicate selectivities by a sampling strategy which is the only feasible way to support predicates with arbitrary data type operations. The optimizer is written in Prolog.

The GUI allows the user to send commands and queries to a kernel and visualize the results. It supports both the executable level language and the SQL level. In the latter case it interacts with the optimizer to obtain a plan (executable query) which it then sends to the kernel. The GUI is extensible by so-called viewers that can offer their own methods to display data types. One of the available viewers (the so-called Hoese-Viewer) allows for a sophisticated representation of spatial data and for animation of spatio-temporal data types. This viewer is itself extensible to support further data types. The GUI is written in Java.

3.4 Basic notations

Each of the tuples in a collection/relation is assumed to have a unique identifier that we need for indexing the contents of the respective attributes. Since we later use a tuple id for a fast access to a certain array slot, and since the tuple ids are not necessarily consecutive in general, we first map them onto a set $\{1, \ldots, n\}$ for the whole computation (the tuple id 0 does not exist; the value 0 is used otherwise). The result of the main algorithm is a list containing the tuple ids of the successfully matched tuples, so we have to map them back to the original tuple ids. In the remainder of this paper, we refer to a tuple id as if they were in consecutive order, beginning with 1. In contrast to tuple ids, the components of a pattern and of the time-dependent attributes start at position 0.

3.5 Pattern language extension

While our previous work only allows for querying sets of symbolic trajectories, in this paper we focus on tuples of numerous time-depending attributes. Similar to the data types *mlabel* (one label per unit) and *mlabels* (arbitrarily many labels per unit), the remaining time-dependent values are realized as sequences of time intervals with an information that belongs to a certain domain. In other words, they can be considered as functions from time into a specific range of values. The time-dependent data types especially supported by our framework are listed in Table 2. Note that a unit of an *mpoint* or *mregion* represents the linear movement of an entity. Curved paths can be approximated by a sufficient number of short units. For more theoretical background, consider [16] (for the data types *mlabel* and *mlabels*) and, particularly, [14].

Previously, a pattern atom was limited to contain temporal information as well as a set of labels. We extended this concept so that constraints can be defined for every time-dependent attribute (having one of the data types from Table 2). Since the length and number of the units is in general different for the time-dependent attributes of a tuple, the user has to determine a so-called main attribute that is essential for the matching. Let m be the number of time-dependent attributes from the tuple description. Hence, analogously to the primary pattern language, every pattern atom a consists of $m + 1$ components $a_0, \ldots, a_m$, where a_0 refers to the temporal data of the main attribute and $a_1, \ldots, a_m$ apply to the time-dependent attributes in the same order as they occur in the tuple description. For example, if a_0 equals `june` at the beginning of the pattern, the main attribute has to start in June for a match, whereas it means no constraint for the remaining attributes that may start earlier. As before, the use of an underscore means that the attribute at the corresponding position is unconstrained. In Table 3 we detail how the components of a pattern atom may be specified for every time-dependent data type.

Note that the spatial data types are *point*, *points*, *line*, *region*, and *rect*, while *int* and *real* are considered as numeric types. In the interval representation, a and b are the left and right limits (real values), and lc and rc indicate whether the interval is leftclosed and/or rightclosed.

Regarding the use of conditions, the previous version was limited to certain information of the respective part of the symbolic trajectory, e.g., the start time, or the set of labels, that could be combined with SECONDO operators. In the pattern defined in Section 3.1, the terms $X.time$ and $Y.time$ refer to the time periods of the sequence of units bound to X and

Table 2 Supported time-dependent data types

Data type	Value domain for each time interval
mlabel	*label*; a character string of arbitrary length
mlabels	*labels*; a set of arbitrarily many *labels*
mpoint	a linear movement from one *point* to another
mregion	a *region*'s linear movement
mbool	a *bool*ean value
mint	an *int*eger value
mreal	a quadratic function of *real*s, or its square root
mstring	*string*; a string of at most 48 characters

Table 3 Pattern atom specification options

Data type	Specification alternatives
mlabel (*s*), *mstring*	name of a DB object of *label* (*s*) type
	character string
mpoint, *mregion*	name of a DB object of a spatial type
mbool	name of a DB object of *bool*ean type
	*bool*ean value
mint, *mreal*	name of a DB object of a numeric type
	*int*eger or *real* value
	interval of the form $\langle a\, b\, lc\, rc \rangle$

Y, respectively (data type *periods*). We extended this concept, so the user may address any of the attributes of the tuple with a term of the form $var.attrname$, in combination with any operator of the SECONDO database system. The binding of variables to sequences of units now refers to units of the main attribute. For example, if Y is bound to [1, 3] as in the mentioned example, and assuming another attribute A with a different fragmentation into units, to evaluate the term $Y.A$ we retrieve the time period of the main attribute corresponding to the units [1, 3] and restrict A to this period. As described in our previous work, we distinguish between easy and complex conditions. While the former refer to only one unit and can be evaluated immediately during the matching process (e.g., `day_of(X.end) = 16`), the latter involve a sequence of units and/or more than one variable and can therefore be verified only at the end of the matching, which holds for the condition from Section 3.1. The other rules of the pattern language remain unchanged and can be obtained from [16] and [36].

3.6 The continuous example

In the following, we define a set of two tuples that will serve as a continuous example throughout this paper. Note that the data set is very small for the sake of brevity. Let t be a tuple (id 1) whose first attribute is the symbolic trajectory from Section 3.1. The second attribute of t is an *mpoint* representing the movement of the person, at the same time of day as the first attribute, i.e., from 19:09:09 until 22:49:09, but with a new unit for every second. Hence, it has not only 4, but $220 * 60 = 13{,}200$ units. In addition, there is an *mint* attribute containing the speed limit of the respective street and therefore having 4 units (value 35 for Wilshire Blvd, 30 for Santa Monica Blvd, 25 for Rodeo Dr, and 30 for Sunset Blvd). We denote the three attributes as Street, Trip, and SpeedLimit. Let t' be another tuple (id 2) that is equal to t except for the time intervals that are all postponed by one month, i.e., the identical movement occurs on January 16, 2016, between 19:09:09 and 22:49:09. We decide to determine Street as main attribute. Finally, let both tuples have an attribute for the person's Name ("John" for t and "Jane" for t') which is not indexed since it is not time-dependent. In addition, let p_0 be the following pattern:

```
X [ [(_ "Camden Dr" _ _) | (_ _ beverlyhills 35)]
    * (2015 "Sunset Blvd" _ <25.0 30.0 t t>)]
// sometimes(speed(X.Trip, wgs1984) > X.SpeedLimit)
```

First, note that p_0 consists of 4 atoms: two in the alternative that is limited by the inner brackets, one wildcard atom and another atom. The outer brackets indicate that the range of the variable X is the whole pattern. Note that `beverlyhills` is a database object of type *region* representing the borders of Beverly Hills, and `wgs1984` is a *geoid* used for computations with geographic coordinates. Semantically, p_0 finds all tuples where the trip starts either at Camden Dr or at any street in Beverly Hills with a speed limit of 35 mph. The trip has to end in 2015 at a segment of Sunset Blvd with a speed limit between 25 and 30 mph, and according to the condition, the velocity must exceed the respective limit at least once. In other words, the pattern finds speeders with a certain movement profile. The **speed** operator converts an *mpoint* into an *mreal*, the comparison with an *mint* yields an *mbool*, and **sometimes** is true if and only if the *mbool* is true in at least one unit.

4 Data structures

The purpose of this section is, first of all, to detail the components of the new index structure for tuples of time-dependent values. In addition, we present auxiliary data structures for the storage of index results and for the course of the exact matching.

4.1 The multi-index

The multi-index is created by the SECONDO operator **bulkloadtupleindex**, taking a database relation and an attribute name, where the type must be time-dependent (cf. Table 2). When the operator is executed, the time-dependent attributes of the relation are processed one after another. We assume that the tuples have $m + c$ attributes, of which m are time-dependent and c are constant ones. Before the operator starts to process the relation, a new index has to be created for each of the m relevant attributes. The type of the new index depends on the data type of the attribute as shown in Table 4.

Every created index is stored in an array, and a mapping that indicates the corresponding index for each attribute, and vice versa, is set up, before the tuples can be processed. For each of the time-dependent attributes, a temporary vector is created, and for each unit of the attribute, the respective value and the exact position (i.e., the tuple identifier and the position of the unit) are appended to the vector.

If the attribute type is *mlabel* or *mstring*, the label/string from each unit is considered. For an *mlabels* attribute, all labels from a unit are inserted. Now we focus on an attribute having the type *mpoint*. From the linear movement inside each unit, we derive the bounding box (i.e., the smallest rectangle completely covering the movement trajectory whose lateral lines are parallel to the coordinate axes) and add it to the vector for the attribute. In case

Table 4 Time-dependent attributes and corresponding index types

Time-dependent data type	Appropriate index type
mlabel, *mlabels*, *mstring*	trie (inverted file) [9]
mpoint, *mregion*	2-dimensional R-tree [17]
mreal, main attribute's time intervals	1-dimensional R-tree
mint	B^+-tree [3, 6]

of an *mreal* attribute, for each unit the minimum and maximum value of the continuous real function are computed. We insert the corresponding interval of real numbers into the respective vector. The units of an *mint* have constant integer values that can be held in the vector. For the main attribute, also the time interval is inserted for each unit (in a separate vector). More precisely, the limits of a time interval (data type *instant*) are transformed into real numbers.

When an attribute is completely processed, the vector is sorted by the value, e.g., by alphabetical order of the labels for an *mlabel*, or by x- and y-coordinates for an *mpoint*. Subsequently, all values from the vector are inserted into the appropriate tree, as listed in Table 4. For each of the different index types, this operation is efficient due to the correct order of the inserted values. Finally, after all time-dependent attributes are processed as described, the multi-index is stored as a persistent database object, having the data type *tupleindex*.

Now we demonstrate how the information from our continuous example from Section 3.6 is inserted into a new multi-index. First, four indexes are created: a trie for the attribute Street, a 2-dimensional R-tree for the geographic position, a B^+-tree for the speed limit, and a 1-dimensional R-tree for the temporal information of the first attribute. Then the attribute Street from tuple t is processed. More precisely, the four street names are inserted into a vector, along with the position of the label, e.g., the pair (1,2) for Rodeo Dr, meaning tuple id 1, unit position 2. We also process the Street attribute from the second tuple, appending the values and positions of the four units (for example, ("Sunset Blvd", 2, 3)) to the vector. Then the vector is sorted by the label (and then by tuple id and by unit position), i.e., its first entry is then ("Rodeo Dr", 1, 2) and its last one is ("Wilshere Blvd", 2, 0). After that, we iterate over the vector and insert all values into the empty trie.

Since Street is the main attribute, the eight time intervals are stored in a vector (after the conversion into intervals of real numbers) and afterwards inserted into the prepared 1-dimensional R-tree. This procedure is more expensive for the attribute Trip for which 26,400 units have to be processed. Their bounding boxes are sorted by x- and y-coordinates and finally stored in the 2-dimensional R-tree. The eight values of the SpeedLimit attribute are inserted into the B^+-tree.

4.2 A container for index results

One possibility of working with an index (or a multi-index, in our case) is to access it every time when it is required. However, as we use an NFA with possible repetitions for the pattern matching algorithm, it is likely that the index is often queried with the same contents, yielding large amounts of repeated results. In addition, in the course of the algorithm, more and more tuples do not have to be considered anymore, due to a successful or unsuccessful matching decision. Hence, an increasing number of results from the multi-index are useless and would cause unnecessary computation cost.

For these reasons, we developed a data structure where all index results are stored, so the multi-index has to be queried only once per atom value. Everytime an atom a is considered for the first time, we store the results for every tuple that is active (what exactly this means is detailed later) in a slot of an array R. Each slot id of R contains the set of unit positions found in the multi-index for the tuple id id. In addition, there are two integer values $pred$ and $succ$ pointing to the previous and successive tuple id that is active and has index results for the contents of this atom. Note that 0 is not a valid tuple id,

so we use the slot $R[0]$ to indicate the first active position $R[0].succ$. For every considered atom a (no matter whether this occurs during the preprocessing or during the exact matching), the array R is computed once and stored into another array $indexResult$ at the position $indexResult[a]$. This structure allows us to insert, retrieve and deactivate contents in constant runtime. Moreover, an iteration over all components with the help of the $pred$ and $succ$ pointers comprises only the relevant (active) ones. If a tuple id i is deactivated during the computation, we have to ensure that an iteration over one of the arrays $indexResult[a]$ ignores the slot $indexResult[a][id]$. Hence, for each atom a, we set the reference $indexResult[a][indexResult[a][id].pred].succ$ of the predecessor of id to its successor $indexResult[a][id].succ$, and vice versa.

Now we explain how the values are written into the slots. Consider an atom a of the form $(a_0\ a_1\ \ldots\ a_m)$. Remember that a_0 represents constraints for the temporal component of the main attribute, while $a_1, \ldots, a_m$ refer to the time-dependent attributes (in the same order as in the tuple description). We loop over all specifications a_i, $i = 0, \ldots, m$. For each a_i, the index results, i.e., a set of pairs of tuple id and unit position, are collected. These unit positions are subsequently transformed into *periods* values with the help of the corresponding time-depending attribute. In constrast to [36] it is necessary to use temporal information for the index results, since the miscellaneous time-dependent attributes may have different temporal distributions, as in our continuous example. As an advantage, instead of numerous unit positions, only one *periods* value is required per tuple. Hence, an index result $I(a_i)$ for a_i has the form $\{(t_1, p_1), \ldots, (t_k, p_k)\}$ where k is the number of tuples having an index result for a_i. Then we compute the intersection $I(a) = \bigcap_{i=0}^{m} I(a_i)$ that also has the mentioned form, since *periods* values are stable with respect to set operations. The result $I(a)$ represents all time periods in which the constraints inside a are fulfilled. Next, we detail how the sets $I(a_i)$ are determined. In general, a specification a_i has the form $r\{a_{i1}, \ldots, a_{il}\}$ where l is the number of components and r is the specified set relation (either omitted for the standard case or superset, intersect, disjoint, equal)[2]. For each a_{ij}, the index results $I(a_{ij})$ are retrieved. According to the set relation, these are either intersected or united (and inverted, if disjoint is chosen), yielding the intermediate result $I(a_i)$. Finally, the set $I(a)$ of index results is transformed into the form $\{(t_1, U_1), \ldots, (t_k, U_k)\}$, where U_i is the set of units of the main attribute during which the specifications from a hold for the tuple t_i.

4.3 A Structure for the exact matching

In order to efficiently file and access the data required for the matching process, we created the IndexMatchInfo data structure, which we call IMI from now on. An IMI instance contains an integer $next$ (for the position of the unit inside the main attribute that is supposed to be matched in the following step), a boolean named $range$ (indicating whether the last considered atom for this IMI instance was a wildcard; if it is true, $next$ or any of its successors may be matched by an atom in the subsequent step; otherwise, $next$ is the only possible match), a mapping $binding$ from a string to a pair of integers (representing the current binding of variables to a sequence of units of the main attribute in case of a pattern with conditions), and a string $prevVar$ (representing the variable considered in the previous update for this IMI instance)[3].

[2]The concept of user-defined set relations was introduced in our previous work, please refer to [36] for details.

[3]Note that in the implementation, variables are represented as integer values for the sake of efficiency. For a better understanding, we chose to use strings in this paper.

The set of IMI instances for one tuple is stored in an array slot whose number represents the tuple id. Again, we apply two references $pred$ and $succ$ for a fast access to every active slot. Since each of these arrays of IMI sets depends on the current NFA state, they are held in an array $indexMatching$, or iM, for short, where the positon represents the NFA state. In the subsequent section, we give a visual example for this structure (Fig. 4).

5 Algorithms

In this section, we introduce the algorithms that are applied to determine the results of the index-supported pattern matching for a pattern p and a set T of tuples with time-dependent values. The suitable SECONDO operator **tmatches** expects a multi-index, a relation, an attribute name (for the main attribute), and a pattern as input parameters. Its output is the stream of tuples that successfully match the pattern. The matching process is divided into two main steps: First, the tuple collection is filtered according to the index results, before the exact matching is performed on the remaining candidates. At the end of the section, we consider computation cost issues.

5.1 Preprocessing

The purpose of this step is to reduce the number of tuples that have to be processed in the main phase.

5.1.1 Crucial transitions

As described in Section 6.1 of [16], a pattern is converted into an NFA, where the trigger of a transition represents an atom. We denote the source and target state and the triggering atom of a transition tr by $tr.source$, $tr.target$, and $tr.trigger$, respectively. Moreover, $\delta(s)$ is defined as the set of transitions outgoing from a state s.

In order to apply the multi-index for the preprocessing, we first identify the transitions that are mandatory for the automaton to arrive at any of its final states, denoted as crucial transitions. The multi-index is queried with the contents of the atoms corresponding to the crucial transitions, and only the tuples whose ids occur in the index results for all crucial transitions/atoms are passed to the exact matching phase. Please refer to [36], Section 4.1.1, for details concerning the determination of the crucial transitions of a pattern. In simple terms, the crucial atoms of a pattern are the non-wildcard atoms that are not located in the branch of a logical alternative.

The NFA corresponding to the pattern p_0 is depicted in Fig. 3. Only atom 3 is crucial. This can easily be derived from the illustration, since the other transitions are either involved in a logical alternative (0 or 1) or in a loop induced by a wildcard (2).

5.1.2 Filtering the set of tuples

Subsequently, we iterate over the set of crucial atoms and apply the indexes. That is, for each non-empty crucial atom a, the array $indexResult[a]$ is filled according to Section 4.2.

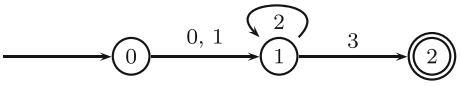

Fig. 3 The NFA computed from p_0

Table 5 Index result for the continuous example

slot / tuple id	0	1	2
$pred$	0	0	–
$succ$	1	0	–
$units$	∅	{3}	∅

At the same time, we update an array $active$ of booleans indicating which tuples are active. For example, if a tuple has index results for a but not for another crucial atom a', the tuple id is removed from the structure $indexResult$, and the value $active[id]$ remains false. The slot $active[id]$ is set to true only if there are index results for every crucial atom. In this phase, false positives cannot be avoided, since no exact matching is performed. However, this method ensures that in the main phase, only tuples whose id occurs in the index results for the crucial pattern atoms are processed, so it remarkably reduces the dataset.

In this step, we only have to consider the final atom of p_0. For the year 2015, the 1-dimensional R-tree yields the result set {(1, 0), (1, 1), (1, 2), (1, 3)}. The search for Sunset Blvd in the corresponding trie produces the results {(1, 3), (2, 3)} (i.e., tuple id 1, unit 3, and tuple id 2, unit 3), and the B^+-tree returns {(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)} when it is queried with the specified real interval. After the transformation into time periods (which could be omitted in this case, but not in general), their intersection, and the backwards transformation into unit positions, we store the unit 3 into the slot $indexResult[3][1]$. Also the $pred$ and $succ$ members are set, e.g., $indexResult[3][0].succ$ equals 1, but $indexResult[3][1].succ$ points to 0, indicating that there is no further active tuple. Finally, $active[1]$ is set to true.

The array $indexResult[3]$ containing the index results induced by the only crucial atom of p_0 is listed in Table 5.

Note that due to the $pred$ and $succ$ variable structure, inactive slots like for the tuple id 2 in the above table are never accessed anymore.

5.2 Exact matching

As stated before, our main challenge was to extend the index-supported pattern matching from accepting only symbolic trajectories to a framework for tuples of several time-dependent values of different data types. Since regular expressions are allowed in the pattern, the number of different paths from the initial NFA state to one of the final states is not limited to the number of final states but may be infinite. Moreover, if the pattern contains conditions, the bindings of variables to unit positions have to be stored, and the underlying database system is used for evaluating the corresponding queries.

The matching of a pattern p with a set T of tuples requires a parallel traversal of a path from the start state to a final state of the NFA for p and of the sequence of units of every time-dependent attribute t_{ij} of every tuple t_i, $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$. Without loss of generality, let t_{i1} be the main attribute of every tuple t_i. However, this traversal is not unique, since the NFA may offer arbitrarily many paths, depending on the pattern and the matching success. Hence, during the process, we always hold the set of active NFA states and have read (conceptually, not physically) an initial subsequence s_{i1} of the units of each main attribute t_{i1}. This sequence can be transferred to all other attributes with the help of

the corresponding time period. Such a state of scanning t_{i1} is exactly represented by an IMI instance, which expresses s_{i1} in terms of the next unit that is to be matched. If the preceding NFA transition was triggered by a wildcard atom, the next unit information is only a lower bound ($range$ = true in this case).

For each tuple, several ways of matching p may exist. Therefore, at any NFA state, for each tuple possibly many IMI objects have to be maintained, each of them representing one possible matching path. Initially, the only active NFA state is 0, and for every tuple t_i that is active after the preprocessing, s_{i1} is empty, so the unit to be matched in the next step is 0.

5.2.1 Initialization

For every tuple t_i that passes the preprocessing, an instance of the structure IMI is created and inserted into the array slot $iM[0][i]$. Remember that in the two-dimensional structure, the first position refers to the NFA state, which is 0 at the beginning. The IMI members are initialized as follows: $next$ is set to 0, $range$ equals false, $binding$ remains empty, as well as the string $prevVar$ (since there is no matching history yet). In case that only wildcard transitions are available at the beginning (e.g., if the pattern starts with a $\star$), $range$ is set to true.

Simultaneously, for each tuple, the global integer value $numActive$ (number of active tuples; initially set to 0) is incremented. This variable is essential for the later matching, since it is decremented for each deactivated tuple, and the algorithm ends as soon as it equals 0. There is also an initially empty list res, where the matching tuple ids will be inserted. The initial state of the structure iM for the continuous example is depicted in Fig. 4. The variable $numActive$ is set to 1.

5.2.2 Traversing the NFA

At the beginning of Algorithm 1, the set of active NFA states is initialized with the initial state 0. Note that for the NFA states as well as for the iM structure, two versions are required; the older ones (iM and S, respectively) are read-only, while new values are stored in the new versions (iM' and S', respectively). At the start of the while loop (lines 5-8), the old versions are replaced by the new ones, and the new versions are cleared. The algorithm collects all available transitions outgoing from any currently active state and stores them in the variable δ_S (line 9). If the set of possible transitions is empty, the computation is aborted and the current result list is returned, since no further matches can be obtained without

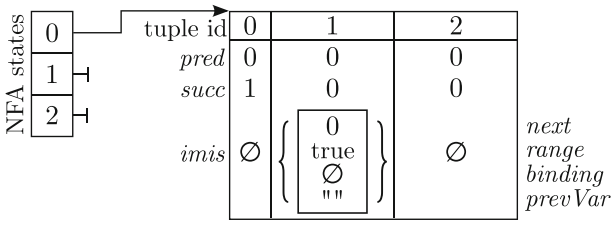

Fig. 4 The initial state of the data structure iM

transitions. Otherwise, a loop over the available transitions is performed, as long as there are still tuples that have not been completely processed.

In contrast to approaches without index support, the time-dependent values are not scanned in a linear way. Instead, Algorithm 1 iterates over the automaton and applies its transitions if the corresponding index results fit the previously saved matching information from iM. The framework of the algorithm is similar to the concept described in [36], however, the inner function $atomMatch$ is completely new.

Algorithm 1: Apply NFA

Input: a pattern p with an NFA including a set Γ of final states;
an non-negative integer $numActive$;
an array $active$ of boolean values;
iM, see Section 4.3;
Output: a list res of tuple ids, initially empty;

```
1  S ← {0};
2  S' ← {0};
3  iM' ← iM;
4  while numActive > 0 do
5      S ← S';
6      S' ← ∅;
7      iM ← iM';
8      clear iM';
9      δ_S ← ⋃_{s∈S} δ(s) ;                          // collect possible transitions
10     if δ_S = ∅ then                    // abort if no transitions are available
11         return res;
12     foreach tr ∈ δ_S do                                   // loop over transitions
13         if tr.target ∉ S then
14             if atomMatch(p, iM, iM', tr, numActive, active, res) then
15                 S' ← S' ∪ {tr.target};
16 return res;
```

For each active state $s \in S$, Algorithm 1 tries to apply every existing transition from all transitions $\delta(s)$ that originate from the state s and whose target state is inactive (for p_0, there are two transitions available at the state 0). In detail, a transition $tr \in \delta(s)$ can be executed if and only if the corresponding pattern atom $tr.trigger$ matches at least one of the IMI instances inside $iM[s]$. This is verified in the function $atomMatch$ that is invoked in line 14. The algorithm stops as soon as there are no active tuples anymore, returning the list with the ids of the successfully matched tuples. If $atomMatch$ returns true, the target state of the transition tr is inserted into the new set S' of active states.

The purpose of the data structure presented in Section 4.3 is to provide all necessary information for each of the possible matching paths. Specifically due to regular expressions that allow steps backwards in the NFA, it is mandatory to enable multiple IMI instances for every NFA state and every active tuple. The structure is crucial for the matching decision in the following way: The tuple t with id id matches the pattern p while a final state s is active if and only if there exists a newly created IMI object which is finished, i.e., whose $range$ value is true or where $next$ equals $|t_{i1}|$, the number of units of the main attribute of t; in other words, where the main attribute is conceptually completely traversed. Finally, we call an instance exhausted if $next$ equals $|t_{i1}|$ during a non-final state, meaning that a match (based on this IMI) is not possible anymore. When the exact matching process deactivates a tuple id id, we set $active[id]$ to false and apply the strategy detailed in Section 4.2.

Algorithm 2: atomMatch

Input: a pattern p with an NFA including a set Γ of final states;
iM and iM', see Section 4.3;
a transition tr, see Section 5.1.1;
an non-negative integer $numActive$;
an array $active$ of boolean values;
a list res;
Output: boolean;

```
1  let newImis be a set of pairs of an IMI instance and a tuple id;
2  if p_tr.trigger is an atom with contents then
3      if ¬indexResult[tr.trigger].isKnown then
4          determineIndexResult(p, tr.trigger) ;                    // see Section 4.2
5      id ← indexResult[tr.trigger][0].succ;
6      while id > 0 do
7          foreach u ∈ indexResult[tr.trigger][id].units do
8              foreach imi ∈ iM[tr.source][id] where imi.suitable(u) do
9                  if match(id, u, p[tr.trigger]) then    // check specs & easy conds
10                     newImis[id].insert(id, (u + 1, false, −, −));
11         pos ← indexResult[tr.trigger][pos].succ;
12 else if p_tr.trigger is an empty atom then
13     id ← iM[tr.source][0].succ;
14     while id > 0 do
15         foreach imi ∈ iM[tr.source][id] do
16             if match(id, imi.next, p[tr.trigger]) then    // check easy conditions
17                 newImis[id].insert(id, (imi.next + 1, false, −, −));
18         id ← iM[tr.source][id].succ;
19 else                                                          // wildcard atom
20     id ← iM[tr.source][0].succ;
21     while id > 0 do
22         foreach imi ∈ iM[tr.source][id] do
23             newImis[id].insert(id, (imi.next + 1, true, −, −));
24         id ← iM[tr.source][id].succ;
25 foreach (id, imi) ∈ newImis do
26     if imi.finished then
27         deactivateId(id);
28         numActive ← numActive − 1;
29         res.append(id);
30     else if ¬imi.exhausted then
31         iM'[tr.target][id].insert(imi);
32 return ¬newImis.isEmpty;
```

In this paragraph, we assume that the considered pattern does not have complex conditions; the other case is detailed in Section 5.2.3. Now we examine the *atomMatch* function, see Algorithm 2. Note that it is invoked with (references to) the pattern p and the arguments iM and iM', i.e., the old and the new version of the structure holding the IMI instances. The old version is used for verifying the matching, and the newly created IMI instances (in the positive case) are inserted into the new version, iM'. The transition parameter tr is important for the access to the correct IMI instances as well as the pattern atom ($tr.source$ and $tr.trigger$). The number $numActive$ of active tuples and the list res are passed for

manipulation. Inside the algorithm, we distinguish between three types of triggering pattern atoms:

An atom with contents. For example, any atom of p_0 except the wildcard atom $\star$. If the index result for p_0 has not been determined yet, this must be done now (line 4). We iterate over the array $indexResult[tr.trigger]$, skipping the inactive tuple ids with the help of the $succ$ member in each slot. For each resulting position (tuple id id and unit u) that is found, it is verified whether there exists a suitable IMI instance in the slot $iM[tr.source][id]$ (line 8). Remember that an IMI instance is suitable if its $next$ member equals u (in case $range$ is false) or $next \leq u$ holds. If there is no suitable instance, we continue with the next index result. Otherwise, and if the attributes at the respective positions are matched by the atom specifications and the easy conditions for the atom are fulfilled (if existing), we create a new IMI instance with $next = u + 1$ and $range =$ false (since the current atom is not a wildcard), see line 10.

Note that a positive index result does not necessarily produce a match in any case. For example, for an *mpoint* attribute, the multi-index yields all unit positions where the bounding box of the movement intersects some specification from an atom, possibly a *region* object. However, the linear movement itself, without the rectangular box, may have no intersection with the region at all. Such expensive verifications have to be performed for each atom with contents, hence for efficiency reasons, we store the respective results in a separate structure for later access, in order to avoid repeated verifications.

An empty atom. That is, `()` or `(_ ... _)`. Accessing $indexResult$ is useless, and we have to loop over the array $iM[tr.source]$, considering all active tuple ids with all existing IMI instances. For each of these instances imi, if the corresponding easy conditions hold, a new instance having $next = imi.next + 1$ and $range = imi.range$ is created (line 17).

A wildcard atom. That is, `+` or $\star$. Again, we cannot access the index results, and all IMI instances from the active tuples inside $iM[tr.source]$ have to be considered. For each existing instance imi, a new one with $next = imi.next + 1$ and $range =$ true is generated, see line 23.

In any of the three cases, after the creation of a new IMI instance, we have to determine whether it is finished or exhausted. The former means that the tuple with id id matches the pattern, so we deactivate the tuple id (i.e., it is removed from the structures described in the Sections 4.2 and 4.3, and the variable $numActive$ is decremented) and append it to the list res. The finished IMI instance is ignored. An exhausted IMI object is also neglected. If none of these two states holds, the new instance is inserted into the set $iM'[tr.target][id]$. Hence, we deduce that Algorithm 1 terminates after no more than $\max\{|t_{i1}| \mid 1 \leq i \leq n\}$ iterations of the while loop, where $|t_{i1}|$ equals the number of units of t_{i1}. However, this worst-case bound only has a theoretical meaning and is realized in very exceptional cases. For the sake of efficiency, the iteration in line 12 of Algorithm 1 is conducted in reverse order, that is, starting with the highest available target state. Causing an earlier decrementation of the value $numActive$, this technique increases the probability of a faster matching decision.

Regarding our example, the first invocation of $atomMatch$ for $tr = (0, 1, 1)$ (meaning the current state 0, atom 1, transition target state 1) queries the indexes with the specifications from atom 1. The index query for atom 1 returns the position (1, 0) (remember that the tuple with id 2 was deactivated), matching the IMI instance in $iM[0][1].imis$, where $next$

is 0. Hence, we create a new instance with $next = 1$ and $range =$ false and insert it into $iM'[1][1].imis$. The state 1 is active now, so the transition (0, 0, 1) pointing to the same state is not considered, and a new iteration of the while loop is started. First, we try to apply the transition (1, 3, 2), but since we obtain only the position (1, 3) from $indexResult[3]$, there is no match with the IMI instance having $next = 1$ and $range =$ false. In contrast, the transition (1, 2, 1) related to the wildcard atom is successful, and a new IMI object with $next = 1$, $range =$ true is inserted into $iM[1][1].imis$. State 1 is still the only active state, and another while loop run begins. The $atomMatch$ function is invoked with the transition (1, 3, 2) again. This time, the retrieved position (1, 3) matches the IMI instance (since $range$ equals true), and we arrive at the final state 2. Since $next$ equals the 4, the number of units of the main attribute, the new IMI instance is finished, so we have a match. Hence, the tuple id 1 is appended to res, $numActive$ is decremented to 0, $active[1]$ is set to false, and res is returned.

5.2.3 Condition processing

If the pattern entails conditions, the current binding of variables has to be kept in each instance of the class IMI. The attribute $binding$ is required for the evaluation of the conditions in SECONDO (see Section 6.3.3 of [16] for details), while $prevVar$ keeps track of the binding variable that was changed most recently, in order to extend the binding correctly. Everytime a new IMI instance is created (except for the initialization step), the attribute $binding$ is copied from the source instance and then updated, depending on the variable $curVar$ of the current atom and on $prevVar$. We consider the following two (non-exclusive) cases:

- If $curVar$ and $prevVar$ have the same non-empty value, the binding of $prevVar$ is extended until $next - 1$, which is the last unit that was matched.
- Now assume $curVar \neq prevVar$. Unless $prevVar$ is empty, we extend the binding of $prevVar$ to $next - 2$ (one unit before the most recently matched one). Moreover, if $curVar$ exists (meaning no contradiction to the previous case), the mapping position $binding(curVar)$ is assigned the pair $(next - 1, next - 1)$.

In any case, the attribute $prevVar$ in the new IMI object is assigned the variable of the current pattern atom. If the IMI object is finished and belongs to a final state (i.e., a match in the version without conditions), the current binding is passed to the condition evaluation. If it fulfills each of the conditions, the IMI instance and the tuple id are processed like in the condition-free version. Otherwise, the IMI object is ignored. For the evaluation, from the binding of variables to intervals of units of the main attribute, we first compute a binding of variables to time intervals. These are required for verifying the conditions that refer to other time-dependent attributes.

The first IMI instance created in the continuous example gets the binding $X \mapsto [0, 0]$, while $prevVar$ is set to X. For the IMI instance emerging from the wildcard transition, the binding remains unchanged. Subsequently, the transition (1, 3, 2) causes the creation of an IMI instance with $next = 4$, so the binding of X is extended to [0, 3], since the considered variable is the same as before. Having arrived at a final state with a finished IMI instance, we proceed to the condition evaluation. Assuming that John's highspeed at Rodeo Dr was actually 45 mph, the evaluation yields true. Hence, the tuple id 1 is appended to res, and $numActive$ is decremented to 0, so the algorithm terminates.

5.3 Runtime considerations

Due to the initialization cost for the arrays $active$, $indexResult$, iM, and iM', the total computation cost for preparing the pattern matching algorithm is certainly linear in the number of tuples and in the number of pattern atoms. The former could theoretically be avoided by using flexible structures instead of fixed-size ones, but the smaller initialization cost would be overcompensated by numerous memory allocations, particularly if the indexes return results for many different tuples. As mentioned before, the maximal number of units of the main attribute is the worst case limit for the number of outer iterations in Algorithm 1. However, instead of more iterations, in most cases the effect of more units in the time-dependent values is that the indexes yield more results that have to be collected and processed. Other influences on the runtime are the selectivity of the pattern, the number, and complexity of conditions (easy or complex, number of applied variables, runtimes of involved operators) as well as the number of time-dependent values in the tuples (and in the pattern). In any case, a certain knowledge about the queried data is helpful to achieve convenient runtimes.

The computation cost for creating the multi-index is in $O(u_T \log u_T)$, where u_T is the total number of units in the tuple collection, due to the required sorting of the temporary vectors, as discussed in Section 4.1. The fact that for huge tuple collections, the vector(s) can possibly not be sorted in the main memory but only on the hard disk may slow down the procedure remarkably.

6 Application example

This section is dedicated to the analysis of a real dataset, in order to demonstrate the expressiveness of our pattern language. The applied dataset contains the mobility traces of 320 taxis in Rome, whose GPS coordinates have been collected for over 30 days [4]. We imported these data into the DBMS SECONDO and obtained 10,278 tuples each of which contains a driver id and an attribute Trip of the data type *mpoint*, representing the geometric trajectory of a driver during one shift, usually having a duration of about 8 hours.

For the geometric trips, we first computed a speed profile and stored it as a new attribute Speed (type *mreal*). Next, we derived the corresponding road names from the OpenStreetMap [27] data of Rome by map matching [25, 31]. In other words, the geometric movement data was matched onto the road network generated from the OpenStreetMap data, resulting in a new attribute Roadname of the type *mlabel* containing the sequence of names of the traversed roads for each of the trips. Note that compared to the raw movement data, the space consumption of this representation is only 14 %. Finally, with the help of elevation raster data from the Shuttle Radar Topography Mission of the U.S. Geological Survey [34], we created altitude profiles for each of the trips as an *mint* attribute named Altitude. Hence, the complete relation for our analysis has the schema

Trip *mpoint*, Roadname *mlabel*, Speed *mreal*, Altitude *mint*.

As a whole, the relation contains 42.4 million units. The multi-index for all these data was successfully created after approximately 7 minutes.

We now search for all trips that match the following pattern:

```
X * (_ _ "Via Giuseppe Zanardelli" _ _) Y *
(2014-02-07-22~2014-02-07 _ "Lungotevere dei Sangallo" _ _)
Z *
```

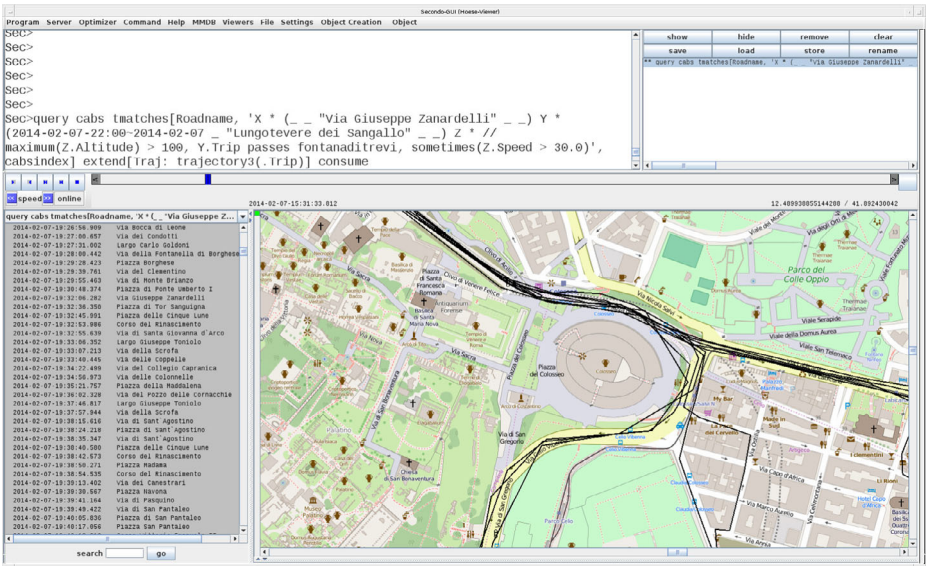

Fig. 5 The trajectories of the matching trips around the Colosseum are depicted on the map. The complete query with the pattern is shown in the *top left* area. The *bottom left* window shows the sequence of road names for one of the trajectories

```
// maximum(Z.Altitude) > 100, Y.Trip passes fontanaditrevi,
sometimes(Z.Speed > 30.0)
```

A trip matches it if it passed the street Lungotevere dei Sangallo between 10 pm and midnight on February 7, 2014, after having traversed Via Giuseppe Zanardelli and if the highest altitude experienced after Via Giuseppe Zanardelli exceeded 100 meters, the Trevi Fountain[4] was passed between the two street traversals, and the speed was higher than 67 mph[5] at least once after Via Giuseppe Zanardelli.

The pattern is matched by seven tuples from the relation, being computed in 0.31 seconds by our implemented approach. A small section of the taxi trajectories around the Colosseum is depicted in Fig. 5, which is a screenshot taken from the SECONDO GUI. Note that the same problem can be solved in 33 seconds without index support.

7 Experimental evaluation

In this section, we first present the evaluation of our proposed model by means of a dataset based upon the BerlinMOD [10] data generator as well as geological data and several patterns. As a baseline for comparison, we apply the pattern matching algorithm without index support. All experiments were carried out on an AMD Phenom II X6 3.3 GHz

[4]The database object `fontanaditrevi` is a small rectangle around the Trevi Fountain.

[5]SECONDO considers the speed of an object in meters per second; 30 m/sec approximately equal 67 mph.

processor with 8 GBytes of main memory, running openSUSE 13.2. From this environment, SECONDO was assigned one processor core and half of the available memory. All runtimes were determined by executing the respective query (operator **tmatches**) five times and taking the median of the last four values, so the duration of the initial loading from the hard disk was ignored. The second subsection compares our work to the approach recently presented in [20].

7.1 Dataset generation and properties

In order to provide a comprehensive evaluation of our work, we had to create a representatively large and scalable dataset. Therefore, we chose to apply the BerlinMOD generator with a scale factor of 1.0, yielding approximately 145,000 non-stationary raw trajectories (a relation with an *mpoint* attribute in SECONDO). In another attribute (type *mlabel*), we added the street name corresponding to the movement with the help of a relation containing the streets of Berlin and a suitable R-tree. Note that the nearest street was determined for every unit of the *mpoint*. However, as consecutive units with the same label are merged into one unit (with an extended time interval), the symbolic representation is very efficient; in this case, the number of units for the Street attribute is reduced by more than 96 % compared to the geometric data. For the third attribute, we first obtained elevation data for Berlin in raster format from the Shuttle Radar Topography Mission of the U.S. Geological Survey [34]. Since SECONDO supports raster data, the data import was straightforward. We combined the imported elevation data with the geographic trajectories, resulting in a time-dependent integer value for each of the latter. Hence, the relation for our experiments has the attributes

Pos *mpoint*, Street *mlabel*, Altitude *mint*.

The geographic trajectories have 56 million units, and with the two remaining attributes, the total number of units in our dataset amounts to 81 million. Hence, the average number of units per tuple is 557. The size of the complete dataset is 17.4 GBytes.

For this evaluation, we created subrelations of the main relation having different numbers of tuples and different numbers of units per tuple. The runtimes for the creation of the corresponding multi-indexes range from one minute for 25.000 tuples to ten minutes for 125.000 tuples.

We applied the following patterns for the evaluation:

$p_1 =$
```
(_ _ "Buckower Damm" <44.0 46.0 t t>) *
```

$p_2 =$
```
* (_ _ "Am Tierpark" _) *
  X [(_ potsdamerplatz _ _) | (12~24 brandenburgertor _ _)] *
  // sometimes(X.Altitude > 40)
```

$p_3 =$
```
X * (2007-06-04~2007-06-13 _ "Leibnizstr." _) Y *
  // maximum(X.Altitude) <= maximum(Y.Altitude)
```

The pattern p_1 finds all tuples starting at the street "Buckower Damm" with an altitude between 44 and 46 meters. For the second pattern, we defined two rectangles `potsdamerplatz` and `brandenburgertor` for the respective places of interest in Berlin. Hence, the pattern p_2 matches the tuples that, after visiting the street "Am Tierpark", pass either Potsdamer Platz or Brandenburger Tor, the latter only between noon and midnight, while the altitude has to exceed 40 meters at least once in both cases. Finally, the third pattern finds the tuples passing Leibnizstr. between June 4 and June 13, 2007, where the maximum altitude experienced before Leibnizstr. does not exceed the maximum altitude afterwards. Note that p_2 has an easy condition and p_3 has a complex condition.

Table 6 Selectivities and runtimes for a growing trajectory collection

# tuples	Selectivity / Runtime (sec.) / Runtime, baseline (sec.)								
	p_1			p_2			p_3		
25,000	0.11 %	0.336	3.732	0.11 %	0.228	34.66	0.23 %	0.735	18.14
50,000	0.08 %	0.67	7.313	0.07 %	0.408	66.67	0.27 %	1.485	36.39
75,000	0.12 %	1.008	10.092	0.05 %	1.009	111.88	0.24 %	2.244	54.09
100,000	0.12 %	1.338	15.462	0.12 %	1.453	150.51	0.21 %	3.069	74.41
125,000	0.12 %	1.669	19.321	0.08 %	1.733	187.88	0.24 %	3.921	93.42

7.2 Evaluation results

The experiments focus on the effects of an increasing number of queried tuples as well as of a growing trajectory size. The corresponding selectivities and the resulting query runtimes are listed in Tables 6 and 7, respectively.

7.2.1 Growing tuple collection sizes

For the first series of experiments, we created five subrelations having between 25,000 and 125,000 tuples that were chosen randomly from the original relation. The number of units per tuple is approximately the same for each dataset. The graphs related to the runtimes of the **tmatches** operator executions with and without index support are depicted in Fig. 6.

As expected, the diagram shows that the computation cost for the index-supported approach is linear in the number of tuples. This is inevitable due to the initialization cost for the arrays $indexResult$, iM, etc., even for patterns with a small selectivity. Also for the version without index support, the runtime is linear. We observe that the index-supported approach outperforms the baseline technique by one or two orders of magnitude, depending on the applied pattern.

The pattern p_1 has no condition, thus no bindings have to be kept and no SECONDO queries need to be evaluated. However, there are up to 2.32 million index results for the interval of altitudes, so that the initialization is expensive for the index-supported version. In contrast, the crucial atom of p_2 causes only up to 3,000 index results, but the subsequent effort is much higher. For the third pattern, we have as much as 46,000 index occurrences for the crucial atom, with the consequence that almost half of the tuples have to be considered for the exact matching algorithm. In addition, p_3 contains a complex condition, so its runtime is above the two others.

Table 7 Selectivities and runtimes for different trajectory sizes

avg. # units	Selectivity / Runtime (sec.) / Runtime, baseline (sec.)								
	p_1			p_2			p_3		
200	0.08 %	0.182	2.918	0.07 %	0.121	26.448	0.23 %	0.353	14.375
400	0.13 %	0.339	2.919	0.16 %	0.233	31.555	0.13 %	0.573	16.177
600	0.12 %	0.425	2.813	0.08 %	0.195	35.986	0.52 %	0.781	17.662
800	0.06 %	0.5	2.81	0.23 %	0.345	39.251	0.2 %	0.938	20.912
1000	0.12 %	0.561	2.878	0.39 %	0.451	42.72	0.14 %	1.038	21.697

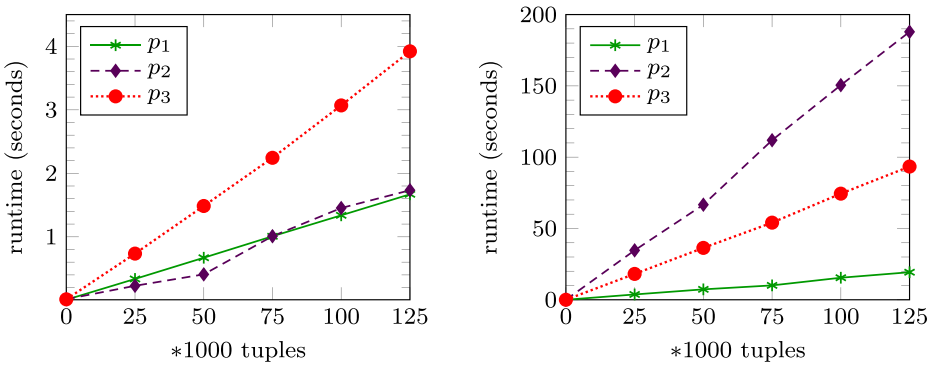

Fig. 6 Runtimes related to a growing number of tuples; index-supported (*left*) vs. non-index-supported (*right*)

Regarding the baseline version without index support, we first observe that the runtimes are between one and two orders of magnitude above the presented approach. The pattern p_1 perfectly suits the non-index-supported approach, since the linear scan ends after the first unit in case of a mismatch, which happens for approximately 99.9 % of the tuples. In contrast, for the other two patterns, more units have to be scanned. The geometric specifications inside p_2 require a spatial operation, i.e., a check whether a segment of a geometric trajectory is enclosed by a rectangle, for a high number of units, thus p_2 causes the highest computation cost.

7.2.2 *Different trajectory sizes*

For the second part of the evaluation, we derived five further subrelations. This time, each of these has a constant number of 20,000 tuples, while the average number of units u_{avg} per tuple ranges from 200 to 1,000. The resulting runtime graphs for the index-supported approach as well as for the baseline version without index support are depicted in Fig. 7.

We immediately observe that the computation cost for the index-supported method is – apparently, in a sublinear way – related to u_{avg}, although there is no algorithmic reason for this connection. This is because in general, the number of index results increases with a higher number of units per tuple. In addition, there are more different ways of matching

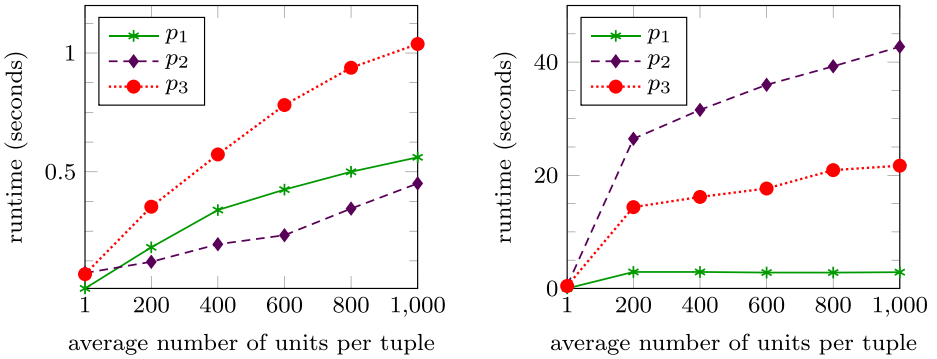

Fig. 7 Runtimes related to different trajectory sizes; index-supported (*left*) vs. non-index-supported (*right*)

a tuple if a value has more occurrences in it, which is more likely for a higher number of units. Note that this does not necessarily result in a higher selectivity. As a whole, the result is similar to the left part of Fig. 6.

For the baseline version, first it is obvious that the runtime for processing p_1 is constant, since the computation in most cases ends after processing the first unit. The graphs for p_2 and p_3 have a linear curve (considered from $u_{avg} = 200$). Again, the index-supported approach outperforms the baseline algorithm by 1-2 orders of magnitude.

7.3 Comparison with another approach

As stated in the related work section, there is no similar approach that can be directly compared to the methods proposed in this paper. We decided to focus on [20] where the authors present a sophisticated index structure (IRWI) and a suitable algorithm for efficiently matching spatio-textual trajectories. We analyzed one of the queries applied for experiments, kindly provided to us by the authors, translated it into our pattern language, and executed the query on a relation of 10,000 BerlinMOD trajectories, including geometric as well as symbolic data (street names), with a space consumption of 750 MBytes. The pattern query comprises four pattern atoms, each having five labels and a large rectangle, alternating with five wildcards, and has a selectivity of 0.1 %. The execution time of 5 seconds is clearly outperformed by the IRWI index [20] consuming only 0.15 seconds.

However, in contrast to our approach, IRWI is specialized for this case and can only handle spatio-textual trajectories. The query language supported by IRWI is limited as the user can only specify a sequence of triples of the form (instant, label set, region) that can match trajectory units occurring in the same order. Further details such as time intervals or spatial objects of other types (e.g., lines or point sets) are not supported, the same holds for regular expression structures, wildcards, or conditions. The approach presented in this paper is able to process any combination of time-dependent attributes, hence it is not customized for a certain case and all results from the multi-index have to be collected once, at least for each crucial atom. Regarding the applied query, the bottleneck is the specification of the four large rectangles (with an area of several square kilometers), forcing the multi-index to consider millions of trajectory units that pass through them. Omitting them reduces the computation time to 0.16 seconds.

Finally, the construction time of the IRWI index for this dataset amounts to 2.7 hours, according to [20]. The multi-index proposed in this paper is built after 18 seconds.

8 Conclusion and future work

Our previous work [16] introduced a comprehensive framework for a general representation of movement in a symbolic space, including an expressive pattern language for symbolic trajectories and a pattern matching algorithm. Subsequently, we proposed a more efficient version of the algorithm based on an index for symbolic trajectories [36].

In this paper, we have presented a new framework for an efficient pattern matching algorithm on tuples of numerous time-dependent values. The framework has been fully implemented in SECONDO. As a major extension of our abovementioned work, we have provided an enhanced pattern language according to the new functionality as well as a data structure consisting of heterogeneous indexes whose number and types depend on the attributes in the set of tuples. With the new language, it is possible to define a pattern for matching not only a set of trajectories but collections of complete tuples of time-dependent values.

The corresponding pattern matching algorithm exploits the multi-index and avoids a linear scan of the attributes. Besides detailing the involved data structures and algorithms, we have also provided an application example with a real dataset and a comprehensive experimental evaluation of our approach, including a comparison with a baseline algorithm without index support.

Future research will create and analyze different distance functions for symbolic trajectories and/or other time-dependent attributes, in order to find similarities and clusters inside movement data. In addition to that, we will consider privacy issues, that is, to what extent a geographic trajectory can be restored from a symbolic one, depending on the type of labels (e.g., street names, activities, transportation modes).

References

1. de Almeida VT, Güting RH, Behr T (2006) Querying moving objects in Secondo. In: MDM, pp 47–51
2. Andrienko GL, Andrienko NV, Heurich M (2011) An event-based conceptual model for context-aware movement analysis. Int J Geogr Inf Sci 25(9):1347–1370
3. Bayer R, McCreight EM (1972) Organization and maintenance of large ordered indices. Acta Inf 1:173–189
4. Bracciale L, Bonola M, Loreti P, Bianchi G, Amici R, Rabuffi A (2014). CRAWDAD dataset roma/taxi (v. 2014-07-17). Downloaded from http://crawdad.org/roma/taxi/20140717
5. Chang JW, Song MS, Um JH (2010) Tmn-tree: New trajectory index structure for moving objects in spatial networks. In: CIT, pp 1633–1638
6. Comer D (1979) Ubiquitous B-Tree. ACM Comput Surv 11(2):121–137
7. Damiani ML, Issa H, Güting RH, Valdés F (2014) Hybrid queries over symbolic and spatial trajectories: A usage scenario. In: MDM, pp 341–344
8. Damiani ML, Issa H, Güting RH, Valdés F (2015) Symbolic trajectories and application challenges. SIGSPATIAL Special 7(1):51–58
9. De La Briandais R (1959) File searching using variable length keys. IRE-AIEE-ACM (Western):295–298
10. Düntgen C, Behr T, Güting RH (2009) BerlinMOD: a benchmark for moving object databases. VLDB J 18(6):1335–1368
11. Erwig M, Güting RH, Schneider M, Vazirgiannis M (1999) Spatio-temporal data types: an approach to modeling and querying moving objects in databases. GeoInformatica 3(3):269–296
12. Forlizzi L, Güting RH, Nardelli E, Schneider M (2000) A data model and data structures for moving objects databases. In: ACM SIGMOD, pp 319–330
13. 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
14. Güting RH, Böhlen MH, Erwig M, Jensen CS, Lorentzos NA, Schneider M, Vazirgiannis M (2000) A foundation for representing and querying moving objects. ACM TODS 25(1):1–42
15. Güting RH, Schneider M (2005) Moving objects databases morgan kaufmann
16. Güting RH, Valdés F, Damiani ML (2015) Symbolic trajectories. ACM TSAS 1(2):7:1–7:51
17. Guttman A (1984) R-trees: A dynamic index structure for spatial searching. In: SIGMOD, pp 47–57
18. Hadjieleftheriou M, Kollios G, Bakalov P, Tsotras VJ (2005) Complex spatio-temporal pattern queries. In: PVLDB, pp 877–888
19. Hopcroft JE, Motwani R, Ullman JD (2001) Introduction to automata theory, languages, and computation - (2. ed.). Addison-wesley series in computer science Addison-Wesley-Longman
20. Issa H, Damiani ML (2016) Efficient access to temporally overlaying spatial and textual trajectories. In: IEEE MDM, pp 262–271
21. (2016). Microsoft: http://research.microsoft.com/en-us/projects/geolife/
22. du Mouza C, Rigaux P (2004) Multi-scale classification of moving objects trajectories. In: Proceedings on SSDBM, pp 307–316
23. du Mouza C, Rigaux P (2005) Mobility patterns. GeoInformatica 9(4):297–319
24. Navarro G, Raffinot M (2002) Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences, Cambridge University Press
25. Newson P, Krumm J (2009) Hidden markov map matching through noise and sparseness. In: ACM SIGSPATIAL. ACM, pp 336–343

26. Nguyen-Dinh L, Aref WG, Mokbel MF (2010) Spatio-temporal access methods: Part 2 (2003 - 2010). IEEE Data Eng Bull 33(2):46–55
27. (2016). OpenStreetMap: http://www.openstreetmap.org
28. Parent C, Spaccapietra S, Renso C, Andrienko GL, Andrienko NV, Bogorny V, Damiani ML, Gkoulalas-Divanis A, de Macêdo JAF, Pelekis N, Theodoridis Y, Yan Z (2013) Semantic trajectories modeling and analysis. ACM Comput Surv 45(4):42
29. Pelekis N, Theodoridis Y (2014) Mobility data management and exploration springer
30. Pfoser D, Jensen CS, Theodoridis Y (2000) Novel approaches in query processing for moving object trajectories. In: VLDB, pp 395–406
31. Quddus MA, Ochieng WY, Noland RB (2007) Current map-matching algorithms for transport applications: State-of-the art and future research directions. Transportation Research Part C: Emerging Technologies 15(5):312–328
32. (2016). Secondo website: http://dna.fernuni-hagen.de/Secondo.html
33. Spaccapietra S, Parent C, Damiani ML, de Macêdo JAF, Porto F, Vangenot C (2008) A conceptual view on trajectories. Data Knowl Eng 65(1):126–146
34. (2016). U.S. Geological Survey: http://srtm.usgs.gov/
35. Valdés F, Damiani ML, Güting RH (2013) Symbolic trajectories in Secondo: Pattern matching and rewriting. In: DASFAA, pp 450–453
36. Valdés F, Güting RH (2014) Index-supported pattern matching on symbolic trajectories. In: ACM SIGSPATIAL, pp 53–62
37. Valdés F, Güting RH, Ossi F (2016) Efficient trajectory analysis for several time-dependent attributes: A case study for roe deer. In: IEEE MDM, pp 337–340
38. Vazirgiannis M, Theodoridis Y, Sellis TK (1998) Spatio-temporal composition and indexing for large multimedia applications. Multimedia Syst 6(4):284–298
39. Vieira MR, Bakalov P, Tsotras VJ (2010) Querying trajectories using flexible patterns. In: Proceedings of the EDBT, pp 406–417
40. Vieira MR, Bakalov P, Tsotras VJ (2011) Flextrack: a system for querying flexible patterns in trajectory databases. In: SSTD, pp 475–480
41. Vlachos M, Gunopulos D, Kollios G (2002) Discovering similar multidimensional trajectories. In: ICDE, pp 673–684
42. Yan Z, Chakraborty D, Parent C, Spaccapietra S, Aberer K (2013) Semantic trajectories: Mobility data computation and annotation. ACM TIST 4(3):49
43. Zhang C, Han J, Shou L, Lu J, La Porta TF (2014) Splitter: Mining fine-grained sequential patterns in semantic trajectories. PVLDB 7(9):769–780
44. Zheng K, Shang S, Yuan NJ, Yang Y (2013) Towards efficient search for activity trajectories. In: ICDE, pp 230–241
45. Zheng Y, Zhou X (eds) (2011) Computing with Spatial Trajectories. Springer

Fabio Valdés is a Ph.D. student at University of Hagen, Germany. He received his Diploma in mathematics from the University of Dortmund in 2011 and subsequently became a research assistant in the group Database Systems for new Applications at the University of Hagen. His major research interests include symbolic trajectories and pattern matching with index support.

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 sys- tems became his major research interests; more recently, also spatio-temporal or moving objects databases. He is a Senior Associate Editor of the ACM Transactions on Spatial Algorithms and Systems and an Editor of GeoInfor- matica. He has previously served as an Associate Editor of the ACM Trans- actions on Database Systems and as an Editor of the VLDB Journal. He is a member of the SSTD Endowment, the organization overseeing the confer- ence series ”Symposium on Spatial and Temporal Databases”, and currently its chair. He has published two German text books on data structures and al- gorithms and on compilers, respectively, and an English text book on moving objects databases, as well as around one hundred journal and conference arti- cles. His group has built prototypes of extensible and spatio-temporal database systems, the Gral system and the SECONDO system.