The VLDB Journal (2019) 28:427–449
https://doi.org/10.1007/s00778-018-0525-6

REGULAR PAPER

A framework for efficient multi-attribute movement data analysis

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

Received: 2 February 2018 / Revised: 1 August 2018 / Accepted: 10 October 2018 / Published online: 31 October 2018

Abstract
In the first two decades of this century, the amount of movement and movement-related data has increased massively, predominantly due to the proliferation of positioning features in ubiquitous devices such as cellphones and automobiles. At the same time, there is a vast number of requirements for managing and analyzing these records for economic, administrative, and private purposes. Since the growth of data quantity outpaces the efficiency development of hardware components, it is necessary to explore innovative methods of extracting information from large sets of movement data. Hence, the management and analysis of such data, also called trajectories, have become a very active research field. In this context, the time-dependent geographic position is only one of arbitrarily many recorded attributes. For several applications processing trajectory (and related) data, it is helpful or even necessary to trace or generate additional time-dependent information, according to the purpose of the evaluation. For example, in the field of aircraft traffic analysis, besides the position of the monitored airplane, also its altitude, the remaining amount of fuel, the temperature, the name of the traversed country and many other parameters that change with time are relevant. Other application domains consider the names of streets, places of interest, or transportation modes which can be recorded during the movement of a person or another entity. In this paper, we present in detail a framework for analyzing large datasets having any number of time-dependent attributes of different types with the help of a pattern language based on regular expression structures. The corresponding matching algorithm uses a collection of different indexes and is divided into a filtering and an exact matching phase. Compared to the previous version of the framework, we have extended the flexibility and expressiveness of the language by changing its semantics. Due to storage adjustments concerning the applied index structures and further optimizations, the efficiency of the matching procedure has been significantly improved. In addition, the user is no longer required to have a deep knowledge of the temporal distribution of the available attributes of the dataset. The expressiveness and efficiency of the novel approach are demonstrated by querying real and synthetic datasets. Our approach has been fully implemented in a DBMS querying environment and is freely available open source software.

Keywords Pattern matching · Multi-attribute data · Indexing

1 Introduction

As a consequence of the recent proliferation of GPS-enabled devices such as cellphones or car navigation systems, a massive and still growing amount of position data is collected every day. In response to this development, researchers have explored new methods for storing, administrating, and analyzing an entity's so-called movement data or geometric trajectory that is represented by the sequence of timestamped geographic positions obtained from the respective device. From a more abstract point of view, the recorded movement data can be considered as a continuous function from time into two-dimensional space, denoted as a moving point [23].

Extracting information from geometric trajectories is often neither efficient nor expedient. Many analysis tasks include finding entities that have passed a certain street, city, point of interest, etc., during some period of time or sequences of such inquiries, instead of focusing on certain geographic coordinates. Hence, a symbolic representation of the movement, in most cases consuming clearly less storage space than the corresponding geometric trajectory because the described property (e.g., the street name or transportation mode) can remain unchanged for longer periods (minutes or

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

Ralf Hartmut Güting
rhg@fernuni-hagen.de

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

even hours) while the position is refreshed more frequently, enables a more convenient and efficient querying, depending on the evaluation purpose. A so-called symbolic trajectory [25] can cover any property of movement or movement-related data and is either directly computed from the raw movement data (e.g., speed categories, cardinal directions) or derived with the help of additional data (for example, altitudes, names of traversed regions or places of interest) or manually entered (such as activities or transportation modes).

For numerous applications, it is useful or even necessary to consider several values that change with time, besides or instead of the geometric movement or a certain symbolic representation. For example, the analysis of aircraft traffic data is more effective if not only the airplanes' position but also further data such as altitude, temperature, or the remaining amount of fuel are recorded at each timestamp. The purpose for analyzing such multi-attribute data can be economic (logistical optimization, customer behavior analysis, targeted advertising), scientific (animal behavior analysis, healthcare), administrative (urban planning, criminal investigation), or private. A comprehensive collection of data types representing values that change over time have been defined as abstract functions, and discrete representations for them have been implemented in the DBMS SECONDO [13,22]. More precisely, each of the time-dependent data types is realized as a chronologically ordered sequence of so-called units that consist of a time interval and a value, for example, the start and end point of a line representing a short segment of a geometric trajectory, or a character string which could be the name of a street, district, activity, etc.

Lately, we presented a framework for pattern matching on datasets of arbitrarily many time-dependent attributes of different types [47] that is based on regular expressions, finite state machines, and a combination of index structures and that can be considered as a precursor of this work. In this paper, we detail a novel framework that also analyzes sets of tuples with any number of time-dependent attributes of different data types, offering several optimizations compared to the mentioned publication. In [47], the user had to assign one attribute whose temporal distribution, i.e., the number and length of its units, was significant for the outcome of the matching algorithm, hence a deep knowledge of the dataset was required, entailing the risk of undesirable results. In this work, this drawback has been removed by processing all time-dependent attributes equally and independently from their temporal distribution. In addition, the user-defined pattern always had to match the complete temporal extent of the data, a limitation that has been removed in the new framework. Finally, the efficiency of the matching algorithm has been improved, mostly by changing the way in which the applied indexes are accessed.

The pattern matching algorithm decides which tuples from a dataset match a certain pattern, where one tuple usually contains the available information related to the movement of a certain entity. The algorithm is divided into two main phases, filtering and exact matching. During the first phase, the tuples for which there are no index results related to certain components of the pattern are pruned. In many cases, this applies to a large number of tuples. The exact matching phase is then performed on the reduced dataset. The pattern is translated into an NFA transition function which is repeatedly executed, while certain information about each active tuple are held and updated inside specialized efficient data structures.

1.1 Insight into pattern language

Next, we provide an insight into the pattern language proposed in this paper. We assume a relation with two attributes, where the first one (named Trip) represents a human trajectory in Los Angeles and the second one (Street) contains the corresponding time-dependent sequence of street names. Consider the pattern[1]

```
(2017 _ "Melrose Av") Z // speed(Z.Trip, wgs) > 20
```

whose three specifications inside the first pair of parentheses are filters for the time and for the attributes Trip and Street, respectively, where the underscore is interpreted as a wildcard for the Trip attribute. Hence, according to the part left of the double slash, all tuples that passed Melrose Avenue in 2017 are found. Due to the variable Z and the specified condition, the resulting tuples are further filtered, and only those exceeding a speed of 20 meters per second at least once after passing Melrose Avenue.

In our previous work [47], the mentioned pattern can be expressed as follows:

```
* (2017 _ "Melrose Av") * Z *
// sometimes(speed(Z.Trip, wgs) > 20)
```

However, its semantics is not equivalent to the pattern above. We now focus on four differences between both approaches, in increasing order of importance:

- The condition must be a boolean expression in [47]. In the current approach, time-dependent boolean expressions are allowed, too.
- The asterisks are not necessary anymore. This is because pattern components now match time periods instead of units (cf. Sect. 3), and attributes do not have to be matched completely.
- In [47], one attribute has to be selected whose partitioning into units is relevant for the matching decision. No

[1] Note that `wgs` is a database object of the type *geoid*, required for precise computations on the earth's surface.

matter which one is chosen, the variable Z from the example above cannot be guaranteed to refer only to the part after Melrose Avenue (due to possible repetitions inside the attribute). In contrast, the current approach ensures that the section represented by Z occurs completely after passing Melrose Avenue.
- The novel approach guarantees that the condition defined in the first pattern has to be evaluated precisely once (or not at all, if the matching fails beforehand). This number can be much larger and even prohibitively large in [47].

1.2 Contributions

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

- We introduce a framework for efficiently analyzing large sets of movement data having any number of time-dependent (movement-related) attributes.
- The corresponding pattern matching algorithm is fully implemented and available as a module of the open source DBMS SECONDO.
- Compared to the previously existing framework, the pattern language and semantics have been changed:
  - The user does not have to select a particular attribute whose temporal distribution is relevant for the matching result; instead, all attributes are now processed equally.
  - We adjusted the semantics of the pattern language (partly as a direct consequence of the previous statement), e.g., time-dependent condition results are now supported.
  - The pattern can not only match the complete temporal extent of the data, but in general any temporal restriction of it. This extends the expressiveness of the pattern language and makes it more suitable for analyzing incomplete data.
  - The matching algorithm is clearly more efficient, mostly due to adjusted indexes and a reduced number of disk accesses.
- One section is dedicated to prove that the novel pattern language is more expressive than its precursor.
- We present an application scenario based on a real dataset.
- The novel approach is evaluated in a series of experiments comprising several patterns of different complexities. A synthetic dataset as well as a real-world dataset (and subrelations of both of them) are applied.
- To our knowledge, this is the only pattern matching approach offering this level of expressiveness and flexibility with respect to the pattern language and possible application domains.

1.3 Paper organization

The remainder of the paper is organized as follows: Related work is discussed in the subsequent section, before Sect. 3 introduces symbolic trajectories and the representation of time-dependent data types in general. An overview of the DBMS SECONDO is also provided. In Sect. 4, we propose our new approach for pattern matching on multi-attribute movement data analysis. The superiority of the novel language compared to its precursor is detailed in Sect. 5. Applied data structures and algorithms are presented in Sect. 6. Section 7 details an application example with aircraft movement data from Aircraft Traffic Control. We provide an experimental evaluation in Sect. 8, before the paper is concluded in Sect. 9 with an outlook to future work.

2 Related work

The approach presented in this paper is connected to several research areas. In the following, we review the state of the art, dividing the research results into three categories: data models and representations for trajectories with semantic information, pattern matching languages, and trajectory indexes. Note that this assignment is subjective and some of the cited publications could also be assigned to another category as they cover more than one.

2.1 Semantic trajectories

Approximately since the beginning of this century, moving objects have become a very active research field [24,58]. The authors of [43] introduce a conceptual trajectory model that regards the movement of an entity as an alternating sequence of stops and moves. Generalized variants [2,37,53] of this model have been proposed subsequently. However, these approaches remain on the conceptual level, such that issues of data management and querying remain unsolved. The model of [1] is based on the concept of stops and moves, too. For a more efficient trajectory analysis, the geometries of semantically important regions are extracted from the movement data. In [3], the authors describe an approach to modeling semantic trajectories as collections of subtrajectories with semantic information such as events, goals or behaviors of a person. However, no specialized query language is provided. A framework for semantically enriching and analyzing trajectories is proposed in [17]. With the help of Linked Open Data collections, for example, geo-referenced social media data, movement data can be annotated with concepts and objects. Converting geo-tagged photographs from social networks into semantic trajectories is explored by the authors of [7]. Additional information such as weather conditions during the travel or type of place of interest are associated

to movement data in order to mine behavior information via cluster analysis. An approach to defining a database model for semantic trajectories relying on the use of data types is outlined in [39].

Orthogonal to the development of generic models, researchers explore efficient access and query processing methods for narrower trajectory classes. The authors of [54] define a semantic trajectory as a sequence of timestamped places, i.e., pairs of a spatial location and a semantic label. This model supports the detection of frequent sequential patterns from a set of semantic trajectories, where a sequential pattern is a sequence of temporally bounded transitions from one group of places to another. The model proposed in [55] represents a trajectory as a sequence of spatial points annotated with a number of activities. Apart from the chronological order of such a sequence, the concept of time is completely disregarded.

In [25], a comprehensive framework for the generalized representation of movement in a symbolic space is presented, including four data types for different kinds of symbolic trajectories. Based on and fully integrated in the well-established data model for moving objects introduced in [23], it is available for moving objects database systems such as SECONDO [13,22] or Hermes [38,39]. The proposed paper is based on its data model.

2.2 Pattern matching languages

A simple symbolic representation can be viewed as a chronologically ordered sequence of symbols. As a consequence, it is possible to apply regular expressions for a pattern matching language on such a representation and hence finite automata [27,31] for a pattern matching algorithm. du Mouza and Rigaux [14,15] are the first to present a pattern language for trajectories defined as sequences of symbols in a discrete symbolic space as well as a corresponding matching algorithm based on a nondeterministic finite automaton (NFA). Such a sequence of symbols denotes the successive zones visited by the considered entity. However, the proposed language does not support precise temporal specifications (apart from the order of symbols) or conditions on variables. Vieira et al. [50,51] propose an expressive pattern language for geometric trajectories allowing the use of variables and conditions. Sequences of timestamped region labels occur only as a data structure within the implementation. The approach is restricted to a partitioning of space into disjoint regions and does not offer a general solution for trajectory annotation. More general geometries are supported in [26], whose authors mainly consider range and nearest neighbor queries without providing a symbolic representation or an expressive query language.

The authors of [8] introduce an ontology-based approach for discovering certain events in trajectories that are enriched with semantics. Status messages of ship containers are processed in order to analyze the routes of the shipped goods and to detect anomalies suggesting fraudulent intentions. Given the container movement data in a suitable format, the presented queries (e.g., find all containers loaded into a vessel that returns to the loading port before reaching its planned destination) could also be realized with the framework presented in this paper. Another contribution in the field of ontology proposes a high-level data model representing trajectory episodes of different granularities [34]. The framework includes several ways of representing spatio-temporal information and supports expressing qualitative and quantitative semantic descriptions.

The framework [25], whose data model for representing moving objects in a symbolic space is already mentioned in Sect. 2.1, also introduces an expressive pattern matching and rewriting language based on regular expressions. It is fully implemented and embedded in the SECONDO implementation of the model of [23]. Related demonstrations analyzing private trajectories of a person and the Microsoft GeoLife [57] dataset are provided in [10,44], respectively. In addition, application challenges are discussed in [11]. An index-supported and thus more efficient version of the pattern matching framework is available [45].

As a major extension of the framework, relations of tuples with arbitrarily many time-dependent attributes of different types can be queried inside one pattern [47]. In other words, the pattern language can not only be applied to symbolic trajectories, but to collections of movement and movement-related data. The approach is efficient due to a composite index and a filtering phase that usually prunes a large part of the data. A related demonstration is conducted in [48]. However, there are several shortcomings such as a limited expressivity of the language, performance issues, and undesirable results in some situations. Further details concerning the work [47] follow in Sect. 3.4. These and other deficiencies have been cured in the proposed approach. For example, the flexibility of its precursor has not only been preserved but even extended, the matching semantics have been changed in order to be more intuitive, and the efficiency of the matching process has been enhanced by approximately one order of magnitude. A demonstration of the novel system has been conducted in [46]. To the best of our knowledge, there is no other approach proposing a fully implemented expressive pattern language that is applicable to datasets containing any number and types of time-dependent attributes.

2.3 Indexing movement data

In order to realize efficient search queries and pattern matching algorithms on trajectory (and related) data, it is mandatory to apply index structures. Numerous publications consider index structures for spatial trajectories, e.g., the 3D R-tree

[49], the TB-tree [40], and the TMN-tree [9]. A survey of indexes for spatio-temporal data of several categories is provided by [33]. In [52], an index structure supporting the detection of similar multidimensional trajectories is detailed.

For efficiently querying collections of symbolic trajectories, a twofold index structure and a suitable pattern matching algorithm are provided [45]. The authors of [28] present an index tailored for the special case of spatio-textual trajectories. Their approach includes a query language that is limited insofar as it does not support regular expressions, conditions, or the use of time intervals.

A sophisticated hybrid index for spatio-temporal-textual data and a corresponding matching algorithm for so-called spatio-temporal keyword patterns in collections of semantic trajectories are proposed in [20,21]. The introduced query language supports spatial, temporal, and textual specifications as well as wildcards and regular expression structures. On the other hand, it does not include variables or conditions, and due to the fixed data model, it is not possible to represent or query further information, such as a numerical attribute for the altitude or different text categories.

The authors of [29] consider the similarity of spatial trajectories fusioned with keywords. A hybrid index structure named ST-tree is presented along with a suitable algorithm that searches for similar trajectories using a probabilistic topic model. Their approach relies on the correctness of the keyword search that may be error-prone, and it seems that the temporal dimension has been completely ignored.

A recommendation framework for activity trajectories is proposed in [56]. It applies a hybrid index structure and a similarity function that detects trajectories according to spatial distance, keyword similarity, and keyword popularity.

The precursor of this work [47] applies a combination of well-known and widespread index structures such as R-tree, B-tree, and trie (inverted file) for its flexible approach, supporting the efficient processing of geometric trajectories, symbolic trajectories, and further time-dependent data such as numeric or boolean values. The tuple id and unit position are stored for every index entry. For efficiency reasons and because our current approach focuses on time intervals instead of unit positions, the indexes store the tuple id and time interval instead.

3 Preliminaries

This section reviews the concept of symbolic trajectories, and, in general, abstract and discrete representations of time-dependent data types. The first two subsections are mostly covered by [25]. Subsequently, an overview over the DBMS SECONDO and basic notations are provided. In the final subsection of this section, we briefly review the results of [47].

3.1 Symbolic trajectories and pattern matching

First, we present a short example of a symbolic trajectory. It describes the sequence of transportation modes used by a person during her/his trip.

[2017-06-13-18:17:11,	2017-06-13-18:21:27)	walk
[2017-06-13-18:21:27,	2017-06-13-18:32:03)	bus
[2017-06-13-18:32:56,	2017-06-13-18:58:40)	train
[2017-06-13-18:59:32,	2017-06-13-19:09:09)	bike

This trajectory is a sequence $U' = \langle u'_1, u'_2, u'_3, u'_4 \rangle$ of four so-called units in chronological order, where each unit 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. The time intervals have to be disjoint but not necessarily continuous (e.g., there is a gap between the third and fourth unit). In the database system SECONDO, the corresponding data type is *moving*(*label*), or *mlabel*, for short. An object of this data type represents a label that changes with time, i.e., a time-dependent character string. The data types *mlabels*, *mplace*, and *mplaces* are closely related to *mlabel* and also introduced in [25]. Each unit of one of them contains a set of labels, a place, and a set of places, respectively, where a place is a label with an additional reference into a repository of geometric information, e.g., river courses or region shapes.

Database systems supporting these (or similar) data types can offer numerous options for querying an object of this kind. For example, in SECONDO there are a large number of operations such as **passes** or **atinstant** that can be applied to extract certain information from a symbolic trajectory. In order to cover more sophisticated queries, a very expressive but still lucid pattern language has been developed as well as a pattern matching algorithm that decides whether a pattern p with m components matches a symbolic trajectory $U = \langle u_1, \ldots, u_n \rangle$. The components of such a pattern can match either one unit of U or a sequence of them, depending on the respective contents of the pattern component p_j and the unit u_i. A component p_j may contain a temporal restriction (e.g., a time interval, a day of the week, or a logical conjunction of such specifications) and/or a set of labels/places. In this context, a match occurs (in the most basic version) if and only if the time interval of u_i is completely covered by the time period specified in p_j and if the labels/places in u_i are also present in p_j. For a match of p and U, there has to exist a partitioning of U into m disjoint subsequences of units (possibly empty), where each subsequence U_j is matched by the pattern component p_j. For example, consider the following pattern

```
({tuesday, 18:00~18:30} "walk") *
(2017-06-13-18:55~2017-06-13-19:15 "bike")
```

which consists of three components. The first one (in parentheses) matches the first unit u'_1 of the symbolic trajectory mentioned above, since the labels coincide and the unit's time interval fulfills both requirements, i.e., it occurs on a Tuesday, between 6 pm and 6:30 pm. Similarly, u'_4 is matched by the final pattern component. The second component, an asterisk, represents a wildcard element that can match any sequence of units. In this case, it matches the sequence $\langle u'_2, u'_3 \rangle$ of remaining units. Hence, the pattern successfully matches the symbolic trajectory U'.

In addition, the pattern language supports regular expression structures as well as variables prepended to pattern components and comma-separated conditions. For example, the pattern

```
X (_ "walk") Y *
Z [(_ "bike") | (_ "motorbike") | (_ "skateboard")]
// get_duration(X.time)<get_duration(Z.time), Y.card=2
```

matches the symbolic trajectory U', too. Note that it does not contain any temporal specifications. The first condition after the double slash compares the temporal durations of the two units bound to the variables X and Z, respectively, while the second one refers to the number of units that belong to the sequence bound to the wildcard variable Y. Any number of conditions may be specified, and every operator and object of the underlying database system can be applied. For example, **get_duration** is a SECONDO operator that computes the duration of a *periods* value. The pattern language for analyzing collections of symbolic trajectories is detailed in [25] and enhanced in [45].

3.2 Representation of time-dependent data types

Based on a comprehensive framework for representing and querying moving objects in databases [16,18,23], in this section we provide an introduction to abstract and discrete representations of time-dependent data types.

The general purpose of that framework is to provide a collection of abstract data types describing moving objects and operations that can be applied to them. For example, the data type *moving*(*point*) (*mpoint*, for short) represents a time-dependent location in the Euclidean plane, while a time-dependent real value is described by the type *mreal*. The operation **trajectory** maps an object of the type *mpoint* to a *line*[2] value, and the **distance** operation returns the time-dependent distance of two *mpoint* values as an *mreal* object.

These and many more data types and operations are integrated into the data model of a DBMS in the following way. The defined data types can be used as attribute types; hence, we may have a relation describing car trips of several persons with the schema

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

The operations can be applied in queries. For example, the query

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

finds pairs of vehicles that eventually have been closer to each other than 100 meters. Note that the **minimum** operation maps the time-dependent distance between the vehicles (data type *mreal*) to a constant *real* value.

From a formal perspective, a system of data types and operations on them can be considered as a (many-sorted) algebra. It consists of a signature providing sorts and operations. Regarding the definition of the semantics, carrier sets and functions have to be assigned to the sorts and operations, respectively.

In the mentioned framework for the representation of moving objects, data types are created from certain basic types and type constructors. The type system itself is described by a signature, where 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 the following signature:

int, *real*, *bool*:		$\rightarrow$	BASE
array:	BASE	$\rightarrow$	ARRAY

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

The type system for moving objects defined in [23] is presented in Table 1. It contains some basic standard types (first line) as well as spatial types (second line). An object of the data type *instant* represents an element from the continuous domain of time, i.e., a time instant or a point of time. For a given static type, the type constructor *moving* provides a corresponding time-dependent type, such as the abovementioned type *mpoint*. The *intime* constructor yields for a static type α a data type whose values are pairs of a time instant and a value of the type α. Finally, the *range* constructor applied to a type α provides another data type whose values are finite sets of disjoint intervals over the domain of α. For example, the mentioned data type *periods* is an abbreviation for *range*(*instant*).

[2] The spatial data type *line* represents (a linear approximation of) a continuous curve in the Euclidean plane.

Table 1 Type system defined in [23]

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

For providing the semantics of the data types, it is necessary to define their domains or carrier sets. In this context, it is important to distinguish between an abstract model and a discrete model, as introduced in [16,18,23]. In an abstract model, the domain of a data type may be defined with the help of infinite sets, for example, regarding the domains of integer or real values or points in the Euclidean plane. Such an abstract model is conceptually simple, but it is in general not directly implementable. In contrast, the possible values of data types in a discrete model have to be defined in terms of finite representations. These can be comfortably mapped to data structures in an implementation.

For a data type α, we denote its carrier set in the abstract model as A_α. When the undefined value $\perp$ belongs to a carrier set A_α, let $\bar{A}_\alpha = A_\alpha \setminus \{\perp\}$ be the carrier set A_α without the undefined value.

Definition 1 Let α be a data type to which the type constructor *moving* is applicable. Then the carrier set of the data type $\underline{moving}(\alpha)$ is defined as

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

In the abstract model, the issue of how such functions can be represented is completely disregarded. The discrete model presented in [18] provides finite representations for all the types of the abstract model. The so-called sliced representation for data types $\underline{moving}(\alpha)$ is introduced. This means that the time domain is partitioned into disjoint time intervals (slices) such that within each slice, the development can be represented by some function of time which is finitely representable. In other words, the function for a slice can be described by a constant number of parameters instead of an infinite set of pairs of a time instant and a value.

The sliced representations of two sample objects of the types *moving*(*real*) and *moving*(*point*) are depicted in Fig. 1. The representation of a single slice, which consists of a time interval and a function description, is called a unit.

For the data types listed in Table 1, a comprehensive set of operations is defined. Most of the operations are generic, i.e., applicable to many of the available data types. Consider the following two operations:

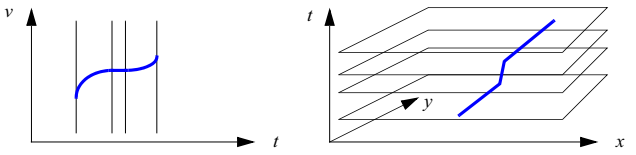


Fig. 1 Sliced representations for the types *moving*(*real*) and *moving*(*point*)

deftime: $\underline{moving}(\alpha) \to \underline{range}(\underline{instant})$
atinstant: $\underline{moving}(\alpha) \times \underline{instant} \to \underline{intime}(\alpha)$

The **deftime** operation returns the set of time intervals during which a moving object is defined, while **atinstant** computes the value of a moving object at a certain time instant. Both operations are generic, since they are applicable to all data types generated by the *moving* type constructor. Please refer to [23] for more details and complete definitions of data types and operations.

3.3 The DBMS SECONDO

In this section, we briefly introduce the DBMS SECONDO, reviewing some of the results of [22]. SECONDO is a prototype DBMS that has been developed at University of Hagen since 1995. It can be run on Linux and MacOS X platforms and is freely available open source software [12]. A clean extensible architecture as well as support for spatial and spatio-temporal data types and applications have always been the main design goals.

The architecture of the SECONDO system is depicted in Fig. 2. It has three major components: the kernel, the optimizer, and the GUI. The kernel does not implement a fixed data model. Instead, it is open for the implementation of a wide variety of DBMS data modules, and it can be extended by algebra modules. More exactly, a particular data model is entirely implemented within such algebra modules. Hence, there are algebras in SECONDO for basic data types, for tuples and relations including operations such as **hashjoin**, for B-trees and R-trees with their build and access operations, for spatial and spatio-temporal data types, and many more. Some algebras are beyond the scope of a relational model and support nested relations, movement in networks, or parallel processing.

The SECONDO kernel evaluates terms over the existing objects and operations. For example, it can evaluate the expression

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

where Trains is a relation containing an attribute Trip of the type *mpoint*, representing trajectories of a metro train. The

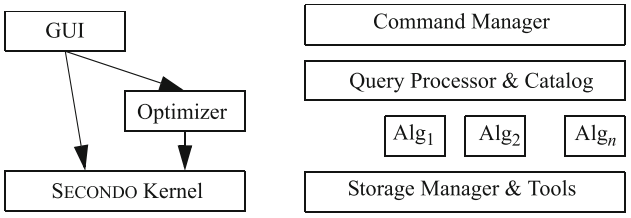


Fig. 2 SECONDO components (left), architecture of kernel system (right) [22]

syntax for an operation can be freely determined. It is often convenient to use postfix notation for query processing operations, e.g., the first argument to the **filter** operation is `Trains feed`. Stream processing is built into the engine. The set of 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 SECONDO optimizer is not as independent from the data model as the kernel. It assumes an object-relational model and supports a language similar to SQL that is mapped to the executable language shown above. The optimizer can be extended by registering types and operations from the executable level and by defining translation rules and cost functions. New index types can be added, and concepts to distinguish between logical and physical indexes are provided. 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 SECONDO GUI sends the user's commands and queries to a kernel and visualizes the results returned by the kernel. It supports the SQL-like optimizer language as well as SECONDO executable language. In the former case, it interacts with the optimizer to obtain an execution plan that is then sent to the kernel. The GUI can be extended by so-called viewers that can realize their own methods of displaying data types. The GUI is written in Java.

3.4 Pattern matching on tuples of time-dependent values

While the framework presented in [25,45] only allows for querying collections of symbolic trajectories, an extension that processes tuples of time-dependent values is proposed in [47]. Similarly to the four types of symbolic trajectories, the remaining time-dependent data types are realized as sequences of units, where a unit consists of a time interval and a finitely representable value. The time-dependent data types supported by the extended framework are listed in Table 2. Consider [25] (regarding symbolic trajectories) and, particularly, [23] for more details.

Table 2 Time-dependent data types supported by [47]

Data type	Value domain for each time interval
mlabel	*label*; a character string of arbitrary length
mlabels	*labels*; a set of arbitrarily many *label* values
mplace	*place*; a label and a reference to a geometry
mplaces	*places*; a set of arbitrarily many *place* values
mpoint	a linear movement from a start to an end point
mregion	a *region*'s linear movement between two points
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

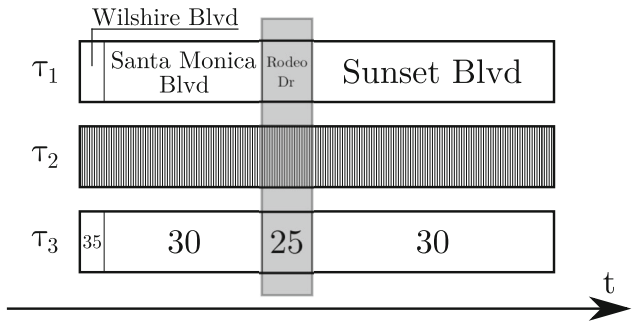


Fig. 3 Visualization of a tuple of three time-dependent attributes; the second tuple unit is highlighted

With the mentioned framework [47], it is possible to formulate sophisticated patterns for analyzing collections of tuples with any number of time-dependent (and constant) attributes. Due to a flexible combination of index structures, the pattern matching process is rather efficient.

A pattern in [47] is only valid in combination with a certain attribute of the dataset which is assigned as main attribute. The choice of this attribute is critical for the matching decision insofar as its partitioning into units is relevant. This is because every component of a pattern can match either one tuple unit or a sequence of them, where a tuple unit is the restriction of a tuple of time-dependent values to the time interval of a unit of the main attribute. The concept of a tuple unit is depicted in Fig. 3, where the example tuple has the schema (τ_1: *mlabel*, τ_2: *mpoint*, τ_3: *mint*). The attribute τ_2 may represent the geometric component of the movement, hence the large number of units is realistic, compared to τ_1 containing the sequence of street names and τ_3, which holds the respective speed limit.

In this figure, we assume that τ is a tuple with three time-dependent attributes from which τ_1 has been selected as main attribute. The highlighted strip covers the tuple unit determined by the second unit of τ_1. As an example of the language of [47], consider the pattern

```
* (_ "Rodeo Dr" _ _) * (_ "Sunset Blvd" _ <28 33>) *
```

Each of the asterisks can match any number of tuple units, including 0. The first component in parentheses matches exactly the tuple unit highlighted in Fig. 3, while the following tuple unit is matched by the pattern's second component in parentheses. Hence, the matching is successful in this example.

However, the approach has several shortcomings, for example:

- The user has to select one time-dependent attribute as main attribute whose temporal distribution into units is critical for the outcome of the matching. Depending on the user's choice, the same pattern may lead to different matching results. If she/he does not know exactly about the temporal distribution, undesirable results are likely, particularly if the attributes have gaps and/or different durations.
- The beginning and end of the pattern have to match the beginning and end of the tuple (more precisely, of the main attribute). This limitation reduces the expressiveness of the pattern language and makes it less suitable for processing incomplete data. Further limitations concern the restricted use of conditions.
- Depending on the pattern, the corresponding NFA may contain loops. This may result in a poor performance, especially for a pattern with conditions that have to be evaluated repeatedly (possibly many times) in this case.
- Due to the implementation of the index structures, retrieving index results causes unnecessary disk accesses. In some cases, the exact matching phase follows multiple matching paths which can be very inefficient, particularly if the main attribute has a large number of units.

As a consequence, in this paper we propose an adjusted pattern language with different semantics that is detailed in Sect. 4. The novel approach does not require the concept of tuple units and a main attribute, instead all the time-dependent attributes are processed with equal priority and based on time intervals instead of units. Asterisks are not required anymore, because temporal gaps between pattern components are implicitly allowed. Moreover, a tuple does not have to be matched in its complete temporal extent. In addition to the data type *bool*, conditions may also be specified as time-dependent expressions of the type *mbool*.

Without reducing the expressivity of the language, the existence of loops in the NFA representing the pattern is avoided. In combination with a reduced number of index accesses and condition evaluations, the performance gain is massive, as we demonstrate in the experimental evaluation (Sect. 8).

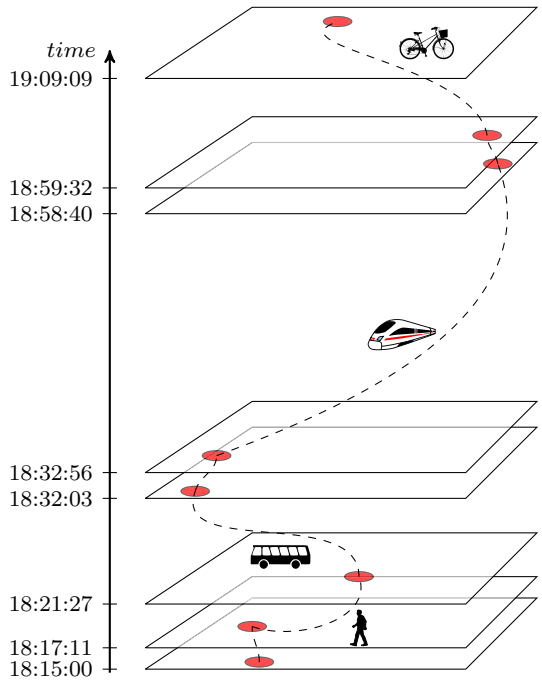


Fig. 4 Visualization of the tuple τ'; transportation mode illustrations gratefully downloaded from [35]

3.5 Running example

The set of two tuples defined in the following is going to serve as a brief running example throughout the remainder of this paper. The notations and their meaning are precisely explained in Sect. 4. Let τ' be a tuple whose first attribute τ'_1 is the symbolic trajectory U' introduced at the beginning of Sect. 3.1, which describes the sequence of transportation modes corresponding to the movement of a person. The second attribute of τ' is assumed to have the type *mpoint* and to represent the geometric movement of this person between 6:15 p.m. and 7:09:09 p.m. in Dortmund, Germany, with a new unit for every five seconds, ending precisely at the city center. Hence, it has not only 4, but $54 * 12 + 2 = 650$ units. We denote the two attributes as TMode (type *mlabel*) and Trip (type *mpoint*).

The development of τ' is depicted in Fig. 4. Let τ'' be another tuple that is almost equal to τ', except for the time intervals that are all postponed by one week, i.e., the identical movement occurred on June 20, 2017. In addition, consider the following pattern[3] p':

```
[(tuesday "aircraft" _)|(_ "walk" dortmund)
 (_ "bus" _)]
X Y (2017-06-13 "bike" _)
// not(X.TMode passes "subway"),
```

[3] Let `dortmund` and `center` be database objects of the types *region* and *point* representing the shape and the city center of Dortmund, respectively.

```
distance(Y.Trip, center)
  <= minimum(distance(X.Trip, center))
```

It matches all tuples that include either a flight on a Tuesday or a walk in Dortmund with a later bus trip. Any of these alternatives has to be followed by a bike trip on June 13, 2017. In addition, both conditions have to be fulfilled. According to the first one, the transportation modes used by the moving entity between the end of the time period π'_1 associated to the first simple pattern element and the period π'_2 of the final element must not include a subway. The second condition is evaluated to true if the distance of the entity to the city center during the period π'_1 is eventually smaller or equal than the minimum distance between π'_1 and π'_2. Note that each of the conditions in p' contains a variable (X and Y, respectively) that also occurs in the first part of the pattern. The result of a condition depends on the so-called binding of the involved variables to time periods.

For easier referencing, we denote the four items in parentheses in the first line as p'_1, p'_2, p'_3, and p'_4 (precise definitions follow in Sect. 4). The composite expression in square brackets is denoted as $[p'_1 \mid p'_2\ p'_3]$.

4 Advanced concepts for multi-attribute movement data analysis

This section is dedicated to the language for pattern matching on tuples of time-dependent values. Of course, there exist similarities with [47], but the new pattern language described in the following has been designed from scratch.

Patterns without conditions are discussed before we proceed toward patterns with conditions. First, two basic definitions are provided.

Definition 2 An *instant* or *time instant* represents a point of time, where time is regarded as linear and continuous, hence it is isomorphic to the real numbers.

As a straightforward consequence of the above definition, comparison operators are applicable to time instants.

Definition 3 Let $\tau = (\tau_1, \ldots, \tau_k)$ be a tuple with k attributes.

1. An attribute τ_j, $1 \leq j \leq k$, is a *time-dependent attribute* if its data type is one of the types listed in Table 2.
2. If τ contains at least one time-dependent attribute, it is called a *tuple of time-dependent values* (or *attributes*).

4.1 Patterns

From now on, let $\tau = (\tau_1, \ldots, \tau_k)$ always be a tuple of time-dependent attributes. For the sake of clarity and without loss of generality, we assume that the time-dependent attributes of τ occupy the positions $1, \ldots, d$ and the constant attributes have the indices $d+1, \ldots, k$, where $1 \leq d \leq k$.

Table 3 Specification alternatives for a pattern atom

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

Definition 4 A *temporal specification* is either a time instant, a time interval, or an infinitely repeated semantic definition such as the name of a weekday, a month, or a time of the day (`"morning"`, `"afternoon"`, `"evening"`, or `"night"`). A time interval can be specified by entering its start and end instant, separated by a tilde. One of the instants may be omitted, which is then interpreted as either "anytime before ..." or "anytime after ...". Abbreviations for intervals are possible, e.g., "2017" and "2017-06-13" (now without tilde) represent the whole year of 2017 and the complete day of June 13, 2017, respectively.

Depending on the data type of the attribute τ_j, for each of the elements of s_j, $j \leq 1 \leq d$, a *specification* may be applied as listed in Table 3.

A tuple of the form $(s_\iota, s_1, \ldots, s_d)$ is denoted as a *pattern atom*, if s_ι is a set of temporal specifications and each s_j, $1 \leq j \leq d$, is a set of specifications suitable for the time-dependent attribute τ_j of τ. Each of the specification sets may be empty (expressed by an underscore), representing a wildcard, as long as at least one of the s_j, $1 \leq j \leq d$, is specified.

Note that the spatial data types are *point*, *points*, *line*, *region*, and *rect*, while *int* and *real* as well as *range*(*int*) and *range*(*real*) are considered as numeric types. The set of textual types consists of *string*, *text*, and *label*(*s*). In the interval representation, a and b are the left and right limits (*instant* values), and the *bool*ean values lc and rc indicate whether the interval is leftclosed and/or rightclosed. Please refer to [25] for a complete list of supported time specifications.

Definition 5 A *simple pattern* is defined as a sequence $\langle p_1, \ldots, p_m \rangle$, $m \in \mathbb{N}$, of *simple pattern elements*, where a simple pattern element is either a pattern atom or a *pattern element*, where the latter has one of the forms $[p]$, $[\bar{p} \mid \hat{p}]$, or $[p]?$, with simple patterns p, $\bar{p}$, and $\hat{p}$.

The meaning of the pattern elements $[\bar{p} \mid \hat{p}]$ and $[p]?$ is related to regular expression structures. Details concerning their semantics follow in Definiton 7.

Regarding the running example (Sect. 3.5), the pattern p' contains two simple pattern elements, where the first one is a pattern element (surrounded by square brackets, with a logical alternative) and the second one is a pattern atom (preceded by the variable Y).

Definition 6 Let π_0 be a time period, that is, $\pi_0 = \langle I_1, \ldots, I_l \rangle$, $l \geq 0$, where each I_j, $1 \leq j \leq l$, is a time interval, and the intervals are disjoint and chronologically ordered. Then the *tuple restriction* $\tau(\pi_0)$ is defined as $(\tau_1(\pi_0), \ldots, \tau_d(\pi_0), \tau_{d+1}, \ldots, \tau_k)$, where $\tau_j(\pi_0)$, $1 \leq j \leq d$, is the restriction of the attribute τ_j during the time period π_0. More precisely, $\tau_j(\pi_0)$ is derived from τ_j by setting it to $\perp$ for all instants outside π_0 and keeping the remaining values.

Note that the restriction of a time-dependent attribute of the type $\underline{m}\alpha$ to a certain time period yields a value of the identical type. For example, the attribute TMode from Sect. 3.5 can be restricted to the period $\langle$[18:45:00, 19:09:00)$\rangle$ with the following result:

[2017-06-13-18:45:00,	2017-06-13-18:58:40)	train
[2017-06-13-18:59:32,	2017-06-13-19:09:00)	bike

Definition 7 Let $\tau(\pi_0)$ be a tuple restriction.

1. A pattern atom $(s_\iota, s_1, \ldots, s_d)$ *matches* $\tau(\pi_0) :\Leftrightarrow \pi_0 \subset s_\iota$ and $\tau_j(\pi_0) \subset s_j$ for every $j \in \{1, \ldots, d\}$.
2. A sequence $\langle p_1, \ldots, p_m \rangle$ of pattern atoms *matches* $\tau(\pi_0) :\Leftrightarrow$ there is a sequence $\langle \pi_1, \ldots, \pi_m \rangle$ of time periods, with $\pi_l \prec \pi_{l+1}$ for every $l \in \{1, \ldots, m-1\}$, such that p_j matches $\tau(\pi_j)$ for every $j \in \{1, \ldots, d\}$. For two non-empty time periods π_1 and π_2, $\pi_1 \prec \pi_2$ holds iff there exist time instants $\iota_1 \in \pi_1$ and $\iota_2 \in \pi_2$ with $\iota_1 < \iota_2$.
3. Let p, $\bar{p}$, and $\hat{p}$ be simple patterns. Then
   - $[p]$ *matches* $\tau(\pi_0) :\Leftrightarrow$ p matches $\tau(\pi_0)$,
   - $[\bar{p} \mid \hat{p}]$ *matches* $\tau(\pi_0) :\Leftrightarrow$ $\bar{p}$ matches $\tau(\pi_0) \vee \hat{p}$ matches $\tau(\pi_0)$,
   - $[p]?$ *matches* $\tau(\pi_0) :\Leftrightarrow$ p matches $\tau(\pi_0) \vee \pi_0 = \varnothing$.

Regarding the running example (Sect. 3.5), p'_1 does not match any of the tuples, p'_2 matches both tuples during the period $\pi'_2 = \langle$[18:17:11, 18:21:27)$\rangle$ (year, month, and day are omitted here), p'_3 matches both tuples during $\pi'_3 = \langle$[18:21:27, 18:32:03)$\rangle$, and p'_4 matches only τ' during the period $\pi'_4 = \langle$[18:59:32, 19:09:09)$\rangle$. Since $\pi'_2 \prec \pi'_3 \prec \pi'_4$ holds, the sequence of simple pattern elements of p' matches the tuple τ' but not τ''.

4.2 Variables

With the help of variables, it is possible to formulate conditions, in order to verify predicates beyond the scope of a simple pattern. For example, the values that a time-dependent attribute assumes at certain time periods (or instants) can be compared, such as the altitude of an airplane or the speed of a car at different stages of the respective journey.

The allowed syntax for a variable is a capital letter followed by any number of letters and/or digits. A variable is associated to a simple pattern element by prepending it, e.g.,

```
X (saturday "bike").
```

If it is written between two simple pattern elements with variables, it refers to the time interval between them, for example, the variable B inside the pattern

```
A (saturday "bike") B C (sunday "train").
```

During a successful pattern matching process, the specified variables are bound to time periods. Hence, the values of all time-dependent attributes as well as temporal properties can be accessed for formulating conditions. Each variable may applied only once in a pattern.

Definition 8 Let V be a domain of possible variable names. Then a *pattern* is a sequence $p = \langle e_1, \ldots, e_m \rangle$ of *pattern elements*, where each e_j, $1 \leq j \leq m$, is either a pair (v_j, p_j) of a variable $v_j \in V$ and a simple pattern element p_j, or just a simple pattern element p_j.

An additional variable may be positioned between two adjacent pattern elements e_j and e_{j+1} (if e_{j+1} has a variable itself), $1 \leq j \leq m-1$, as well as in front of e_1 (if e_1 has a variable) and behind e_m.

For a pattern p, the corresponding simple pattern $\langle p_1, \ldots, p_m \rangle$ is denoted as $simple(p)$.

Definition 9 Let $p = \langle e_1, \ldots, e_m \rangle$ be a pattern with $simple(p) = \langle p_1, \ldots, p_m \rangle$. Then p *matches* τ *with binding* $B :\Leftrightarrow$ there exist m time periods $\pi_1, \ldots, \pi_m$, with $\pi_j \prec \pi_{j+1}$ for $1 \leq j \leq m-1$, such that p_j matches $\tau(\pi_j)$ for each $j \in \{1, \ldots, m\}$. The corresponding binding is

$$B = B_{elem} \cup B_{inter} = \bigcup_{i=1}^{m} b_i \cup \bigcup_{j=1}^{m+1} b'_j, \text{ where}$$

$$b_i = \begin{cases} \{(v_i, \pi_i)\} & \text{if } e_i = (v_i, p_i) \\ \varnothing & \text{if } e_i = p_i, \end{cases}$$

$$b'_j = \begin{cases} \{(v'_j, \pi'_j)\} & \text{if } \exists v'_j \text{ after } e_{j-1} \text{ and/or before } e_j \\ \varnothing & \text{else.} \end{cases}$$

In this context, if p matches τ, each time period π_i in the set B_{elem} is the maximum time period in which the pattern

element e_i matches the tuple τ. A time period π'_j of B_{inter} is determined as follows[4]:

$$\pi'_j = \begin{cases} [\pi_{j-1}.start, \pi_j.end] \setminus (\pi_{j-1} \cup \pi_j) & \text{if} 1 < j \leq m \\ \{(-\infty, \pi_1.start)\} & \text{if} j = 1 \\ \{(\pi_m.end, \infty)\} & \text{if} j = m+1 \end{cases}$$

Note that the variable v'_1 (if existing) is located in front of the first simple pattern element e_1. Similarly, the position of v'_{m+1} is behind the final simple pattern element e_m.

As mentioned before, after a variable is bound to a time period, certain properties can be accessed via attributes of the variable.

Definition 10 Let B be a binding and let (v, π) an element of B. Then the terms $v.time$, $v.start$, $v.end$, $v.lc$, and $v.rc$ refer to the time period π, its initial instant, its final instant, and the leftclosed/rightclosed flags of π, respectively. Moreover, the notion $v.name(\tau_j)$ can be applied to access the value $\tau_j(\pi)$, $1 \leq j \leq k$. Note that $v.name(\tau_j)$ has the same data type as the original attribute τ_j.

Regarding the running example and applying the time periods introduced at the end of Sect. 4.1, the accessible properties for one possible binding of the variables X and Y are listed in Table 4. The values refer to the only matching tuple τ', where we omitted the year, month, and day information.

Table 4 Accessible properties of the variables X and Y for the running example

Expression	Data type	Value
$X.time$	*periods*	{[18:32:03, 18:59:32)}
$X.start$	*instant*	18:32:03
$X.end$	*instant*	18:59:32
$X.lc$	*bool*	true
$X.rc$	*bool*	false
$X.TMode$	*mlabel*	⟨([18:32:03, 18:32:56), ⊥), ([18:32:56, 18:58:40), "train"), ([18:58:40, 18:59:32), ⊥)⟩
$X.Trip$	*mpoint*	units from 18:32:03 to 18:59:32, i.e., ⟨. . . , ([18:36:08, 18:36:13), ((7.59, 51.81), (7.54, 51.63))), . . . ⟩
$Y.time$	*periods*	{[18:59:32, 19:09:09)}
$Y.start$	*instant*	18:59:32
$Y.end$	*instant*	19:09:09
$Y.lc$	*bool*	true
$Y.rc$	*bool*	false
$Y.TMode$	*mlabel*	⟨([18:59:32, 19:09:09), "bike")⟩
$Y.Trip$	*mpoint*	units from 18:59:32 to 19:09:09, i.e., ⟨. . . , ([19:09:02, 19:09:07), ((7.47, 51.52), (7.46, 51.53))), . . . ⟩
$TMode$	*mlabel*	the complete TMode attribute
$Trip$	*mpoint*	the complete Trip attribute

4.3 Patterns with conditions

With the help of variables, additional conditions can be formulated for a pattern. A condition is an expression over attributes of the applied variables (Definition 10), constant values, and database objects that can use any operation available on the respective data types. The result type of a condition must be either boolean, i.e., it can be evaluated to be either true or false, or *mbool*, which means that the result of the condition depends on the instant of evaluation and can be true or false (if the *mbool* value is not defined at the evaluation instant, we consider the result as false). The general syntax of a pattern with conditions is as follows:

```
<pattern with variables> // <cond. 1>, ..., <cond. c>
```

For a successful matching result, all boolean conditions have to be fulfilled, and the conditions of the type *mbool* must have a common non-empty time period where all of them are evaluated to true.

Regarding the first condition of p', the restriction of the attribute TMode to the period mentioned in Table 4 (lines 6-8) never assumes the value "subway"; thus the expression is evaluated to true. The second condition is also fulfilled, since the expression on the left of the comparison operator equals 0. Hence, p' matches τ' with the mentioned binding of the variables X and Y.

5 Language comparison

In this section, we compare the pattern language presented in this paper, denoted as P, and its precursor, denoted as Q [47], with respect to their expressive power. Before the general proof, we provide a brief example showing that a pattern in P can express a query that cannot be realized in Q.

Let $\tau'' = (\tau''_1, \tau''_2)$ be a tuple of time-dependent attributes, where τ''_1 has the type *mlabel* and assumes the constant label value "Hyde Park" between 10 and 11 a.m. (dates are omitted for the sake of brevity), otherwise it is undefined. The second attribute describes a geometric trajectory (type *mpoint*)

[4] The leftclosed/rightclosed flags displayed here may vary: In the first case, the interval is lc/rc iff π_{j-1}/π_j is lc/rc; in the second case, the interval is rc iff π_1 is not lc; in the third case, the interval is lc iff π_m is not rc.

moving inside London and is defined only between 11:30 and 11:45 a.m. Consider the pattern[5]

```
(_ "Hyde Park" _) (_ _ london)
```

which is an element of P. It matches the tuple τ'', because the two specifications can be matched in the given order. However, in Q it is not possible to define a pattern that can match τ''. This is because a main attribute must be selected whose time intervals are relevant for the matching decision. This means, if τ_1'' is chosen, the first component matches the only unit from 10 to 11 a.m., but the matching algorithm of [47] stops because the main attribute is completely traversed while the pattern (or the NFA, respectively) is not. Selecting τ_2'' as main attribute results in a mismatch at the first pattern component whose corresponding time interval (10 to 11 a.m., from τ_1'') does not correspond to the definition time of τ_2'' from 11:30 to 11:45 a.m.

Theorem 1 *For the pattern languages P and Q, the property $P \supsetneq Q$ holds.*

Proof We first show that all pattern semantics existing in Q can also be expressed in P. Let $q \in Q$ be a pattern. As a major difference to a pattern $p \in P$, the pattern atoms $q_1, \ldots, q_m$ of q are strictly related to units (or sequences of units) of the main attribute τ_1, and the start of q_1 (end of q_m) can only match the beginning of the first unit (end of the final unit) of τ_1. This behavior can also be achieved with p. The initial (final) instant of a time-dependent attribute τ_i can be accessed in a condition, e.g., `X.start = inst(initial(Trip))` or `Z.end = inst(final(Trip))`. Moreover, every pattern atom of p can be limited to match one time interval (instead of a sequence of two or more) by applying a condition of the form `no_components(deftime(Y.time)) = 1`.

The wildcard atoms $*$ and $+$ from Q have been omitted in P. However, the asterisk is implicitly replaced because in P we assume that between two pattern atoms there can always be a temporal gap, whose properties can still be accessed with the help of a variable (as in Q). If two atoms are desired to match consecutive time intervals (i.e., in Q no wildcard is located between them), a condition such as `X.end = Y.start` can be applied. The wildcard $+$, matching at least one unit of the main attribute in Q, can be replaced in P by defining the conditions `Y.start = inst(initial(getunit(Trip, getPosition(Trip, Y.start))))`, restricting the time period associated to `Y` to start at the beginning of a unit of the `Trip` attribute, as well as its analogy `Y.end = inst(final(getunit(Trip, getPosition(Trip, Y.end))))`.

[5] We assume that `london` is a database object representing the shape of the city of London.

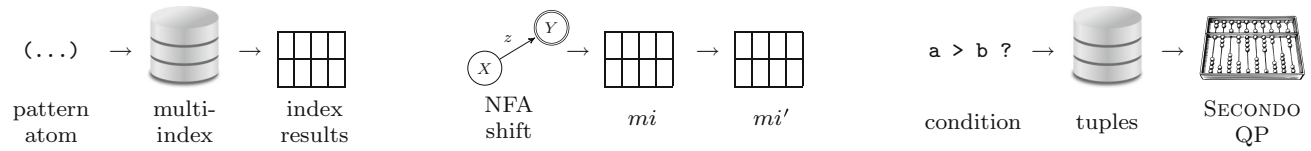
All other items of the pattern language Q, including alternatives, optional elements, and combined specifications, can be directly applied in P, too. As seen in the previous paragraph, it is rather inconvenient to express certain structures from Q in P. This is because the language P presented in this paper does not focus on the distribution of the time-dependent attributes into units. However, with some knowledge of available SECONDO operations, a translation into P can be found. Hence, $P \supset Q$ holds.

On the other hand, there are patterns in P whose semantics cannot be expressed in Q. A pattern atom p_j of $p \in P$ matches the tuple τ during a time period that in general consists of more than one interval, depending on the index results for p_j. Repetitions can also be realized in Q with the help of regular expression structures. Nevertheless, every single matching in Q is still strictly related to the units of τ_1, limiting the expressiveness of the language. Now we consider a time-dependent attribute τ_i that is assumed to have a larger temporal extent than τ_1, more precisely, it starts earlier and ends later. In this situation, in Q it is impossible to access the parts of τ_i occurring before the start of τ_1 and after the end of τ_1, because every pattern in Q is limited to match precisely the temporal extent of the main attribute. In contrast, a pattern atom in P may be applied to match a particular set of attributes, no matter whether other attributes are defined during that time period or not. Moreover, in P conditions with a time-dependent result can be specified, e.g., `speed(X.Trip) > 25.0`, having a result of the data type *mbool*. Conditions in Q must have the result type *bool*. This proves $P \neq Q$, and with $P \supset Q$ from above, $P \supsetneq Q$ follows. □

6 Implementation concepts and details

In the following, we present the data structures and algorithms applied for realizing a flexible and efficient pattern matching on tuples of time-dependent values. More precisely, given a set $T = \{\tau^{(1)}, \ldots, \tau^{(n)}\}$, with $n \in \mathbb{N}$, of such tuples (all having the same schema) and a pattern p, the objective is to determine a subset T' of T containing exactly the tuples that match p.

We assume that each of the n tuples in a collection/relation has a unique identifier which is needed for indexing the contents of the respective attributes. Since the tuple ids are used for efficiently accessing certain array slots, and since the tuple ids are not necessarily consecutive in general, we first map them onto a set $\{1, \ldots, n\}$ during the whole computation. After the list of the ids of the successfully matches tuples is returned by the main algorithm, the list entries are mapped 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.



(a) Preprocessing (Filtering Step) **(b)** Matching (NFA traversal) **(c)** Condition Evaluation

Fig. 5 Overview of the matching process, divided into three main stages; some illustrations gratefully downloaded from [35]. **a** Preprocessing (Filtering Step), **b** Matching (NFA traversal) and **c** Condition Evaluation

Since SECONDO is a relational DBMS, the tuples of time-dependent values are organized in a relation which is completely scanned for constructing the multi-index. When the index is accessed with a certain specification from the pattern, it internally returns a sequence of tuple ids and time intervals. However, it is not necessary to access the relation, unless the pattern contains one or more conditions.

6.1 Overview of the matching process

Before presenting the applied data structures and algorithms in detail, in the following we give an overview of the matching process that is illustrated in Fig. 5.

In the preprocessing phase (Sect. 6.3.1), depicted in Fig. 5a, the multi-index (cf. Sect. 6.2.2) is queried with the contents of a pattern atom (e.g., a street name or a geometric location). The results are inserted into the index result container which is introduced in Sect. 6.2.3.

Figure 5b shows the matching phase that is detailed in Sect. 6.3.2. An NFA transition, which is equivalent to a pattern atom, requires updating every suitable instance of the class IndexMatchInfo (introduced in Sect. 6.2.4). The instances from the previous iteration are located in mi, while the updated ones are inserted into mi'.

Finally, the evaluation of conditions (Sect. 6.3.3) is illustrated in Fig. 5c. Variable expressions (for example, `X.Trip`, referring to an attribute) are replaced by the corresponding data retrieved from the persistent relation. The transformed expression is then passed to the SECONDO query processor that executes it and returns the result.

The multi-index and the original relation are stored as persistent database objects. All other structures only exist in the main memory.

6.2 Data structures

This section introduces the most important data structures applied during the matching process.

6.2.1 NFA

In one of the first steps of the computation, the user-defined pattern is translated into an NFA whose transitions represent exactly the instant patterns. If one of the final states is active—there may be more than one final state due to regular expression structures—the matching process can be considered as successfully finished (apart from verifying the additional conditions). The NFA depicted in Fig. 6 corresponds to the running example pattern defined in Sect. 3.5. Note that the alternative $[p'_1 \mid p'_2\ p'_3]$ is translated into two different paths from state 1 to state 3.

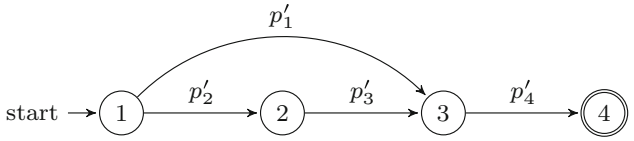


Fig. 6 The NFA corresponding to the running example pattern

Table 5 Time-dependent attributes and corresponding single index types in a multi-index

Time-dependent data type	Appropriate index type
mlabel(*s*), *mplace*(*s*), *mstring*	trie (inverted file)
mpoint, *mregion*	2-dimensional R-tree
mreal	1-dimensional R-tree
mint	B^+-tree

6.2.2 Multi-index

In [47], a composite index structure for sets of tuples of time-dependent values is detailed. In this section we review some of the previous results and focus on the novelties developed since then. A multi-index is an index structure deployed for the efficient pattern matching approach described in this paper. For each of the time-dependent attributes of the dataset, the corresponding multi-index contains one single index of a suitable type. The type of the single index depends on the attribute type as listed in Table 5.

When a multi-index for a dataset is created, first a new single index is created for each time-dependent attribute. For an efficient insertion of all values, the attributes are processed one after another. More precisely, the processing of an attribute is divided into three steps:

1. For each tuple, the value of the attribute is read, and for every scanned unit three items are stored in a vector: tuple id, time interval, value representation (e.g., a label in case of an *mlabel* attribute, or a rectangle representing the bounding box of a segment of an *mpoint*).
2. The vector is sorted by the value representation (for example, by alphabetical order of the labels or by x- and y-coordinate of the bounding boxes).
3. The entries of the sorted vector are bulkloaded into a single index of the corresponding type.

As a crucial difference to the cited work, we store the time intervals corresponding to the respective values instead of unit positions inside a time-dependent attribute. In the previous version, for every position retrieved from the multi-index, it was necessary to access the corresponding attribute inside the tuple and then to read the time interval at the respective position. In the novel approach, the time intervals are directly retrieved from the multi-index. The reason for this change is twofold: First, the new method is more efficient because no access to the persistently stored tuples is required, and second, considering the proposed pattern semantics focusing on time periods instead of units and their positions, it is consequent to store the time intervals directly. Section 6.3 details how the index results are processed. In the experimental evaluation (Sect. 8), we provide an efficiency comparison of both approaches. In addition, there is no separate index for the time intervals of a certain attribute anymore. This is because all attributes are treated equally, as stated in the previous sections.

The multi-index corresponding to the running example (Sect. 3.5) contains a trie for the labels of the TMode attribute and a two-dimensional R-tree holding the bounding boxes of the Trip attribute. We depict the trie in Fig. 7, where each leaf node contains pairs of a tuple id (which can only be 1 or 2 in our example) and a time interval (I'_j and I''_j represent the time intervals of the units of the TMode attribute for the tuples τ' and τ'', $1 \leq j \leq 4$).

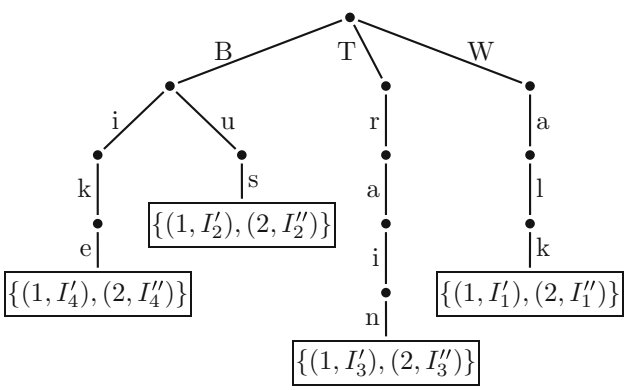


Fig. 7 The trie corresponding to the running example

6.2.3 Index result container

In order to avoid multiple accesses to the same contents of the multi-index, which may be likely due to possible repetitions inside the NFA, we store all retrieved index results in a specialized data structure with constant insertion, retrieval, and deletion cost. Another reason for deploying such a container is that more and more tuples are not considered anymore during the matching algorithm, resulting in an increasing number of useless index results when accessing the multi-index directly.

The applied structure consists of a two-dimensional array whose dimensions represent the pattern atoms and the tuple ids, respectively. The value in each array slot (i, j), $1 \leq i \leq m$, $1 \leq j \leq n$, has the type *periods* and corresponds to the time period in which the specifications inside the pattern atom at position i hold for the tuple j, based on the retrievals from the multi-index. In addition, every slot contains references to its (vertical) successor and predecessor (set to 0 if not existing), which is particularly helpful for efficiently pruning tuple ids and for fast scans over all index results for a pattern element. When a value is inserted or deleted, the references are updated immediately.

Note that a similar concept has been introduced in [47], where sets of unit positions are stored instead of time periods. In some situations, the structure described here consumes more storage space because a time interval consists of two instants (8 Bytes each) and two boolean value (1 Byte each), compared to 4 Bytes for a unit position. However, sometimes one label occurs several times in a row, so that the corresponding time intervals can be combined to a single one.

In Fig. 8, we present the described data structure for the index results of the running example, where m equals 4 and n equals 2. Note that for the sake of clarity, the year, month, and day information as well as the references to predecessors are omitted. How to compute the values is detailed in Sect. 6.3.

6.2.4 Exact matching support

While the auxiliary structure presented in the preceding section is motivated exclusively by efficiency reasons, we now introduce a data structure that is crucial for the exact matching

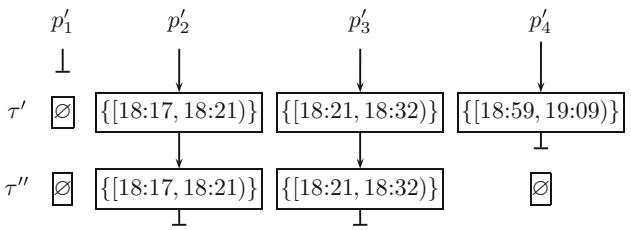


Fig. 8 The index result container for the running example; time instants occur on 2017-06-13 for τ' and on 2017-06-20 for τ'' (seconds omitted for the sake of brevity)

process. The IndexMatchInfo (IMI) structure encapsulates a partial binding of pattern elements to time periods, modeled as a mapping from integers (representing the pattern elements) to *periods* values. In contrast to the index result container whose entries are closely related to pattern atoms (which can be considered as equivalent to NFA transitions), the IMI instances represent the state of the matching process for a certain NFA state and a particular tuple. Again, references to the successive and preceding tuple having IMI instances are deployed for inserting, deleting, and accessing the respective information efficiently. The whole structure is denoted as MatchInfo, and in the following, we denote a particular slot referring to an IMI instance as $mi(s, t)$, where s and t represent an NFA state and a tuple id, respectively.

A similar concept has been detailed in [47]. However, in our novel approach, the IMI data structure is simpler and more efficient insofar as it exclusively stores the binding of pattern elements to time periods. In addition, no more than one IMI instance is required per state and per tuple id because the NFA does not contain repetitions anymore.

For example, we focus on the final NFA state during the matching process of the pattern p' and the tuples τ' and τ'' (Sect. 3.5). After p'_4 matches τ' during the period {[2017-06-13-18:59:32, 2017-06-13-19:09:09)}, the binding of the two pattern elements $[p'_1 \mid p'_2\ p'_3]$ and p'_4 is recorded in a new IMI instance. The created object is placed at the slot $mi(4, 1)$, referring to the fourth NFA state and the first tuple. The remaining slots of the structure are empty.

6.3 Algorithms

The pattern matching algorithm is divided into a filtering phase and an exact matching phase. Both of them are realized differently compared to the approach detailed in [47]. The efficiency of our technique is discussed at the end of this section.

6.3.1 Filtering phase

After the pattern is translated into an NFA, we apply the latter to exclude as many tuples as possible from further computation. This is done as follows:

We first determine the so-called *crucial transitions*, defined in [47] as the set of NFA transitions that have to be executed for a successful traversal, i.e., for a path from the start state to any of the final states. Essentially, these are all transitions that are not part of a logical alternative. For example, the NFA depicted in Fig. 6 has only one crucial transition, because the transition corresponding to the pattern atom p'_4 is the only segment that cannot be substituted by other transitions. In the next step, we determine all index results for the specifications existing in the pattern atoms corresponding to the crucial transitions and insert them into the container structure described in Sect. 6.2.3. More precisely, for every non-empty specification set, a time period is retrieved from the multi-index for every tuple having a result (by uniting the results for every single specification). Their intersection is computed and then again intersected with the specified temporal restriction s_t. For example, the label "bike" from p'_4 yields the respective *periods* value $\{I'_4\}$ for τ' and $\{I''_4\}$ for τ'' from the trie (illustrated in Fig. 7). These intermediate results are then intersected with each other (separately for each tuple) and then with the temporal specification 2017-06-13 of p'_4, with the result $\{I'_4\}$ for τ' which is inserted into the index result container, and an empty time period for τ''. If a certain tuple has no index result for one of the crucial transitions, the latter cannot be executed for that tuple, thus there is no possibility for the pattern to match it as the transition is mandatory for a successful traversal. Hence, tuples without index results for a crucial transition are not considered anymore, such as τ'' in the running example. We can even exclude all tuples that do not have index results for every crucial transition, usually a large part of the dataset.

In our previous work [47], the multi-index as well as the index result container store the tuple id and unit position for every value that exists in the dataset. Regarding the access times it is more efficient to store *periods* values throughout the multi-index, since the previous approach required additional hard disk accesses for retrieving the time interval that corresponds to each unit position and combining them to a time period. The same argument is the reason for storing *periods* values in the index result container, too.

In case the NFA contains at least one crucial transition, for each tuple that passes the preprocessing one IMI instance with an initially empty binding is created and inserted into the slot $mi(0, i)$ (for the tuple id i). For each of the pattern atoms corresponding to a crucial transition, the retrieved index results are directly stored in the index result container. Otherwise, if there is no crucial transition, no tuples can be pruned and such an instance has to be deployed for every tuple of the dataset. Simultaneously, the global integer variable $numActTup$ (initially set to 0) is incremented for every tuple id with an entry in mi, describing the number of tuples that are currently involved in the computation and therefore matching candidates. The exact matching algorithm stops as soon as this variable equals 0. If a tuple is successfully matched by the pattern, its id is appended to a list named $result$ which is initially empty.

6.3.2 Traversal of the NFA

The general idea for the exact matching algorithm is to execute transitions of the NFA and to store and update information about the matching state and the current binding for each active tuple, in order to avoid expensive scans of the time-dependent attributes. Despite the semantic changes

compared to [47], the applied concept remains similar, as listed in Algorithm 1. Throughout this section, we denote the set of transitions of an NFA as δ and the set of transitions outgoing from a state s as $\delta(s)$. For a transition $tr \in \delta$, its source state, transition number (equivalent to the number of the pattern atom), and destination state are denoted as $tr.src$, $tr.atom$, and $tr.dest$, respectively.

Algorithm 1: applyNFA

Input: a pattern p including an NFA with s states, transition function δ and a set φ of final states;
a non-negative integer $numActTup$;
a vector $isActive$ of boolean values;
mi, see Section 6.2.4;
Output: a list $result$ of tuple ids, initially empty;

```
1  let S ⟵ {0}, S' ⟵ {0}, mi' ⟵ mi;
2  while numActTup > 0 do
3      S ⟵ S';
4      S' ⟵ ∅;
5      mi ⟵ mi';
6      clear mi';
7      δ(S) ⟵ ⋃_{s∈S} δ(s);          // collect available transitions
8      if δ(S) = ∅ then return result;
9      foreach tr ∈ δ(S) do    // loop over transitions
10         if atomMatch(mi, mi', tr, numActTup, isActive, result)
           then S' ⟵ S' ∪ {tr.target}
11     foreach id : isActive[id] ∧ mi'[j][id] = ∅ ∀ 0 ≤ j ≤ s do
12         deactivate(id);   numActTup ⟵ numActTup − 1;
13 return result;
```

When Algorithm 1 is invoked, the set S of active NFA states is initialized with state 0. For the set of active states and for the mi data structure, two versions are required: in each iteration of the while loop, S and mi are read-only, i.e., inside the atomMatch function, both are applied as information source. On the other hand, S' and mi', being cleared at the beginning of every iteration (lines 4 and 6), are enriched with new data when the atomMatch function is executed. More precisely, NFA states that become active due to a successful transition are inserted into S' and newly created IMI instances are placed into a slot of mi'. When the successive iteration begins, S and mi are cleared and then receive the contents of S' and mi', respectively. This technique ensures that an NFA state becoming active in a particular iteration cannot be the source of a transition simultaneously. Similarly, a newly created IMI instance must not be used instantaneously but in the subsequent iteration of the while loop.

For every active state $s \in S$ and every transition that originates from it, the atomMatch function is invoked. The target state of a transition becomes active if and only of a true result is returned. We list the atomMatch function in Algorithm 2. Subsequently, the ids of all tuples still being active but without any newly created IMI instance (e.g., due to a negative result of the condition evaluation) are deactivated (lines 11 and 12).

Algorithm 2: atomMatch

Input: a pattern p including an NFA with s states, transition function δ and a set φ of final states;
mi and mi', see Section 6.2.4;
a transition tr;
a non-negative integer $numActTup$;
a vector $isActive$ of boolean values;
a list $result$ of integers;
Output: boolean;

```
1  if indexResult[tr.atom] = ∅ then queryIndex(p, tr.atom);
2  id ⟵ indexResult[tr.atom][0].succ;
3  while id > 0 do    // loop over tuples with index results
4      foreach imi ∈ mi[tr.src][id] :
5          imi.lastBinding ≺ indexResult[tr.atom][id] do
6          if tr.dest ∈ φ then
7              deactivate(id);   result.append(id);
8              numActTup ⟵ numActTup − 1;
9          else mi'[tr.dest][id].insert
10             (imi ∪
               {(elem(tr.atom), indexResult[tr.atom][id])})
11     id ⟵ indexResult[tr.atom][id].succ;
12 return ¬mi'.isEmpty;
```

Note that the version of the algorithm proposed here is applied for the case of a pattern without conditions. The other (more complex) scenario is detailed in Sect. 6.3.3. If the index result for the atom $tr.atom$ has not yet been determined (neither in the preprocessing phase nor in an earlier atomMatch invocation), this has to be done at the beginning. We consider all tuples with a non-empty index result for $tr.atom$ and a suitable IMI instance, where such an instance is regarded as suitable if there are time instants ι_{imi} inside the most recently added time period of the current binding and $\iota_{index} \in indexResult[tr.atom][id]$ with $\iota_{imi} \prec \iota_{index}$, abbreviated as $imi.lastBinding \prec indexResult[tr.atom][id]$.

In case one of the final states becomes active (line 6), i.e., the pattern is successfully traversed, a complete match of the pattern and the currently considered tuple is achieved, so the latter can be deactivated and appended to the result list. The function $deactivate$ removes all existing IMI instances (from mi and mi') as well as the entries in the index result container related to the forwarded tuple id. Otherwise (line 10), a new IMI instance is created with the binding from the previous instance imi extended by the time period retrieved from the index result container and associated to the current pattern element (denoted by $elem(tr.atom)$).

6.3.3 Matching with conditions

If the considered pattern contains conditions, the bindings are applied to compute their results. In order to complete

an existing binding, we apply Definition 9 and associate the respective time periods to the existing intermediate variables (this step is performed after the command in line 10). Also if a final state becomes active, the binding has to be completed (line 6). After that, we pass it to the condition evaluation function that initially replaces all expressions of the form $v.\alpha$ (see Definition 10; for example, `X.TMode, Y.Trip, A.time`) by the corresponding value, depending on the applied binding and the currently considered tuple. The adjusted condition is then transferred to the SECONDO query processor that executes it and returns the result whose data type is either *bool* or *mbool*. For example, the first condition of the pattern p' has the result type *bool*, because the **passes** operation has a constant result. On the other hand, the second condition compares a time-dependent value (the **distance** operation applied to an *mpoint* value and a spatial object yields a result of the type *mreal*) to a constant one (the **minimum** operation computes the constant minimum of an *mreal* value), hence its result type is *mbool*. We extract the time periods with a true value from the time-dependent condition results and compute their intersection. If the latter is empty or if at least one of the constant conditions is evaluated to false, the set of conditions is not fulfilled. In this case, the atomMatch function continues with the next available IMI instance. Otherwise, if all conditions are fulfilled, a complete match of the respective tuple is detected, and the commands from line 7 are invoked.

6.3.4 Complexity discussion

In the following, we discuss the complexity of the presented algorithms. Note that for one pattern matching query, Algorithm 1 is executed exactly once. It then invokes Algorithm 2 for every available transition, if there are still active tuples. In contrast to [47], where repeated negative condition evaluation results could cause severe performance issues, the novel atomMatch algorithm has to be run only once per NFA transition. Note that the while loop condition serves only as an additional stop criterion in order to avoid unnecessary computations for the case that no tuple is active anymore.

The complexity of Algorithm 2 itself primarily depends on whether the multi-index has already been queried with the current pattern atom. If not, this step is executed in line 1, finding and storing all units of all attributes of all involved tuples in the worst case. By all means, every index result temporarily stored in $indexResult$ has to be processed for every imi instance that exists for the current transition and tuple id. The maximum number of these instances equals the number of alternative NFA paths from state $tr.src$, e.g., there are two different options in state 1 of Fig. 6.

Imagine the case of 100,000 trajectories with three attributes (types *mpoint*, *mlabel*, *mint*), each of which contains 1000, 100, and 200 units, respectively, for every tuple. For the sake of clarity, we assume that there are no repetitions inside a tuple, for example, no street name occurs twice in a trajectory. Let p be a pattern with four atoms, each with one specification for every attribute, no alternatives, having a reasonable selectivity of 1%. Then the atomMatch function is invoked four times. For each of these calls, an R-tree, a trie, and a B-tree are queried, producing 10, 1, and 2 results, respectively, requiring the while loop to iterate over 13 tuples.

As a conclusion, the total algorithm is inefficient in the worst case. However, in a realistic scenario with regard to the collection of trajectories and the choice of the pattern, the complexity is reasonable. Most of the actual runtime is consumed for accessing the persistent index structures.

7 Application example

In this section, we present an application example from the field of aircraft traffic analysis. The objective of Aircraft Traffic Control (ATC) systems is to optimize air traffic concerning safety, efficiency, and environmental protection by tracking and analyzing the movement of aircraft. The ATC dataset considered in the following contains more than one million timestamped aircraft position and altitude recordings from one week in the French airspace in April 2008, resulting in approximately 53,000 flights after the import into SECONDO. In a further step, we computed an attribute for the cardinal direction, expressed in the form "W", "SW", "S", etc., and another one for the names of the traversed regions, e.g., "Bretagne" or "Alsace". While the former attribute was derived from the geometric trajectories in a straightforward way, we determined the latter with the help of a relation containing the names and shapes of all French regions. Hence, the schema of the complete relation is

Flights (Id: *int*, Pos: *mpoint*, Altitude: *mreal*, Course: *mlabel*, Region: *mlabel*).

With the help of this dataset, we will now explain the semantic differences between the previous approach and the work described in this paper. Consider the following pattern:

```
X (15:23~ _ <10300 10500> "NW" "Corse") Y
Z (_ parc _ "N" "Lorraine")
// X.Altitude >= maximum(Z.Altitude)
   get_duration(Y.time) > 50 * oneMinute
```

It matches all flights that traversed the region Corsica in northwestern direction with an altitude between 10.3 and 10.5 kilometers after 3:23 pm, and later overflew the spatial object `parc`[6] in the Lorraine region heading North. The flight altitude above Corsica is required to be greater or equal than the maximum altitude above Lorraine at least once. Finally,

[6] The spatial region object `parc` describes the shape of the Parc Naturel Régional de Lorraine.

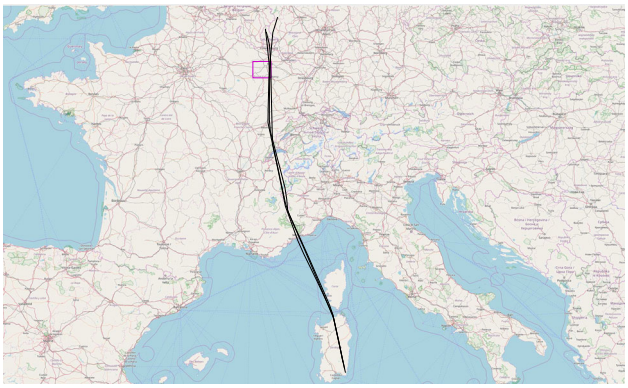


Fig. 9 The trajectories of the three flights matched by the pattern

the time spent between both regions has to exceed a duration of 50 minutes.

This explanation is valid for the novel framework, and there are three tuples that are matches by the pattern. However, if the previous version is executed with the same pattern (extended by wildcards required in that version to indicate possible gaps), no matching tuple is found. The main reason for this behavior is that the old version's matching decision is based on the units of the attribute chosen by the user. For example, if the user selects Region as main attribute, the specifications in every pattern atom must hold for the duration of a unit with the contents "Corse" and "Lorraine", respectively, which is at least 6 minutes. However, the region object `parc` (second pattern atom) is overflown in approximately 2 minutes, so it is impossible for this atom to match a tuple during such a long unit.

In contrast, the new approach is much more flexible insofar as its matching decisions are independent from the temporal distributions. It simply determines the time period during which the specified properties hold. The three flights matched by the abovementioned pattern are depicted in Fig. 9. Note that the object `parc` is represented by a purple rectangle.

8 Experimental evaluation

This section provides an efficiency evaluation of the presented framework as well as a comparison between our previous and current approach. We applied a synthetic dataset as well as a real dataset of taxi cabs in Rome.

All runtimes mentioned in this section were achieved on an AMD Ryzen 7 1700 8-core computer with a main memory of 32 GBytes, running openSUSE 42.2. From these resources, SECONDO has been assigned one processor core and half of the available memory.

8.1 Brinkhoff dataset

With the help of the freely available Brinkhoff network-based generator for moving objects [5,6], we created 100,000 geometric trajectories in the administrative district Arnsberg in Germany. The durations and lengths of the trajectories range from 5 seconds and 14 meters to 51 minutes and 176 kilometers, with average values of 12 minutes and 33 kilometers, respectively.

We applied the Brinkhoff generator with the help of the district's shape file obtained from the Geofabrik website [19]. The created trajectories were imported into a new SECONDO relation with 100,000 *mpoint* objects. In order to add further time-dependent attributes to the dataset, we first constructed the road network of the Arnsberg district inside SECONDO using OpenStreetMap [36] data provided by [19] and a publicly available SECONDO script.[7] Subsequently, we computed the sequence of streets passed by each trajectory via map matching [32,41], represented by a symbolic trajectory of the type *mlabel* for each tuple. For the third attribute, we downloaded and imported the elevation raster data [30] for the Arnsberg district and then determined the time-dependent altitude (type *mint*) for every trajectory. Finally, with the help of the geographic region data of all counties of the administrative district (including independent cities) and a corresponding R-tree, we added an attribute describing for every tuple the sequence of visited counties, for example, Dortmund, Hagen, or Märkischer Kreis. Hence, the schema of the complete relation is as follows:

Brinkhoff (Id: *int*, Trip: *mpoint*, Roadname: *mlabel*, Altitude: *mint*, County: *mlabel*)

The relation contains 68 million units (over all attributes) and occupies almost 8 GBytes of disk space.

We compared the performance of the **tmatches2** operator that has been introduced in this paper to its precursor (operator **tmatches**). Both operators were executed with increasing numbers of tuples as well as different patterns. We prepared the set of experiments by creating four subrelations of 20,000, 40,000, 60,000, and 80,000 tuples from the original relation, all with the same average tuple size of approximately 698 units per tuple, in order to eliminate the possible influence of varying tuple sizes. The creation of the multi-indexes for the five relations consumed between 50 seconds for 20,000 tuples and 4 minutes for 100,000 tuples. Hence, the fact that time intervals instead of unit positions are stored in the new version of the multi-index does not have a noticeable effect on the construction time.

We applied the subsequent patterns for the evaluation (slightly adjusted in order to have similar semantics in the previous version):

[7] The script `OrderedRelationGraphFromFullOSMImport.SEC` is located in the directory `secondo/bin/Scripts`.

```
p = [(evening _ "Westfalendamm" _ "Dortmund") |
     (19:35~22:00 _ _ _ "Unna")]
q = X (2016-07-01-19:30~ _ _ _ {"Dortmund", "Hagen"}) Y
    Z (_ _ _ <430 440> {"Hochsauerlandkreis",
                        "Märkischer Kreis"})
    // not(Y.Roadname passes "Westfalendamm")
r = A B (2016-07-01-19:00~ _ "Wittbräucker Straße" _
                                              "Dortmund")
    C D (_ _ _ _ {"Olpe", "Siegen-Wittgenstein"}) E
    // C.Trip passes ruhr, D.start-B.end < 30*oneMinute
```

For example, the pattern r matches all entities passing the street Wittbräucker Straße in Dortmund after July 1st, 2016, 7 pm. Later, the trip has to include at least one of the counties Olpe or Siegen-Wittgenstein, after having traversed the river Ruhr (represented by the database object `ruhr` of the type *line*), according to the first condition. In addition, the time spent between Wittbräucker Straße in Dortmund and one of the other counties must remain below half an hour.

Both variants of the pattern matching operation were executed with all three patterns. Each of the runtimes was determined by starting the corresponding query four times and computing the median of the resulting runtimes. The runtimes and selectivities of all queries are listed in Tables 6 and 7. The runtime graphs are depicted in Fig. 10.

In Fig. 10, we observe that all runtime graphs are approximately linear in the number of tuples, which is due to the initialization and processing cost arising for each (active) tuple as well as to the number of index results that increases with a growing dataset. The performance gain for the new version of the framework is a little more than one order of magnitude, regarding the curves corresponding to the patterns p and r, although the selectivities are lower for the previous version. Concerning the graphs for q, the performance advantage is clearly smaller. However, according to the selectivities, the number of successfully processed tuples is between 20 and 40 times larger. The reason for the large selectivity differences is described in Sect. 7. If there are more tuples in the result set, the cost for processing them increases. Hence, considering the huge difference between the selectivities, the benefit in the performance for q must be valued higher than only the runtime advantage.

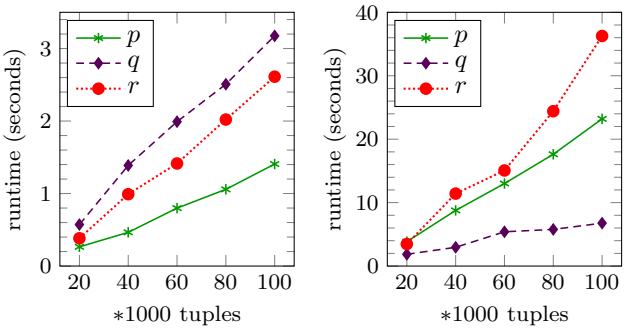


Fig. 10 Runtimes related to a growing number of tuples: new (left) and previous (right) version of the framework

Table 6 Novel approach: selectivities and runtimes for an increasing number of tuples

# tuples	*p*		*q*		*r*	
	sel.	Time	sel.	Time	sel.	Time
20,000	20.08%	0.265	0.52%	0.572	0.11%	0.383
40,000	14.49%	0.465	0.53%	1.388	0.14%	0.991
60,000	16.55%	0.799	0.51%	1.991	0.13%	1.415
80,000	14.05%	1.058	0.4%	2.508	0.14%	2.021
100,000	12.05%	1.407	0.39%	3.175	0.17%	2.612

Table 7 Baseline approach: selectivities and runtimes for an increasing number of tuples

# tuples	*p*		*q*		*r*	
	sel.	Time	sel.	Time	sel.	Time
20,000	19.11%	3.84	0.02%	1.856	0.08%	3.47
40,000	13.09%	8.77	0.02%	2.955	0.08%	11.42
60,000	15.01%	13.01	0.02%	5.401	0.08%	15.07
80,000	12.73%	17.62	0.02%	5.775	0.09%	24.41
100,000	10.88%	23.20	0.01%	6.765	0.1%	36.27

8.2 Taxi dataset

The dataset considered in the following is based on the mobility traces of 320 taxis in the city of Rome, whose timestamped positions have been recorded for one month [4]. We imported them into SECONDO and obtained 162,000 geometric trajectories (type *mpoint*). Similarly to Sect. 8.1, we applied map matching with OpenStreetMap data in order to create an *mlabel* attribute containing the sequence of street names corresponding to the movement. In addition, we downloaded the geometries and names of the 15 districts of Rome [42] and imported them as a SECONDO relation. As a consequence, we were able to add a further attribute of the type *mlabel* describing in which of the districts the respective taxi was located at a certain time. The extended relation has the following schema:

Cabs (CabId: *int*, TripId: *int*, Trip: *mpoint*, Roadname: *mlabel*, District: *mlabel*)

It contains 17.6 million units (over all attributes), occupying 2.5 GBytes of disk space. Similarly to the previous section, we compared the performances of the **tmatches** and **tmatches2** operators with respect to a growing num-

ber of tuples, ranging from 32,000 to 160,000. The runtimes for creating the corresponding multi-indexes amounted to 16 seconds for the smallest collection and 72 seconds for the 160,000 tuples relation. The datasets were queried with the following patterns:

```
s = (2014-02-02~ _ "Lungotevere dei Sangallo" _)
    (_ _ "Corso Vittorio Emanuele II" _) (_ vatican _ _)
t = X (morning _ _ {"Arvalia Portuense",
                    "San Giovanni / Cinecitta"}) Y
    Z [(_ vatican _ _) | (_ _ _ {"Centro Storico",
                "Parioli / Nomentano - San Lorenzo"})]
    // avg_speed(X.Trip, wgs) > avg_speed(Y.Trip, wgs)
u = A B (~10:00 _ _ "Appia Antica") C
    D (morning _ {"Via Giuseppe Zanardelli",
              "Via del Plebiscito"} "Centro Storico") E
    // hour_of(E.end) < 12, not(E.Trip passes tiber)
```

Note that `vatican` in pattern t is a *region* object representing the Vatican's boundary. The SECONDO operator **avg_speed** computes the average speed of an *mpoint* object, in this case, the Trip attribute. Regarding the pattern u, `tiber` corresponds to the course of the river Tiber (type *line*). Hence, the pattern u matches all tuples passing the district Appia Antica before 10 am, followed by one of the streets Via Giuseppe Zanardelli and Via del Plebiscito in the Centro Storico district, still in the morning. Also the end of the trajectory has to occur before noon according to the first condition, and finally we require that the Tiber river is not traversed during the last section of the movement (after Centro Storico).

The novel pattern matching operation (**tmatches2**) as well as the previous one were executed with each of these patterns. We list the runtimes and selectivities of all queries in Tables 8 and 9. The runtime graphs are depicted in Fig. 11.

From these results we can conclude that the pattern matching framework introduced in this paper outperforms its precursor by almost an order of magnitude. As mentioned (Sect. 8.1), this difference would be even larger without the selectivity gaps.

Table 8 Novel approach: selectivities and runtimes for an increasing number of tuples

# tuples	*s*		*t*		*u*	
	sel.	Time	sel.	Time	sel.	Time
32,000	0.05%	0.179	0.83%	0.717	0.14%	3.61
64,000	0.04%	0.442	0.82%	1.602	0.15%	8.53
96,000	0.05%	0.806	0.87%	2.582	0.15%	11.34
128,000	0.06%	1.281	0.93%	3.675	0.15%	14.61
160,000	0.05%	1.703	0.91%	4.777	0.13%	18.57

Table 9 Baseline approach: selectivities and runtimes for an increasing number of tuples

# tuples	*s*		*t*		*u*	
	sel.	Time	sel.	Time	sel.	Time
32,000	0.04%	1.226	0.36%	4.34	0.05%	21.11
64,000	0.03%	2.603	0.33%	11.68	0.04%	38.55
96,000	0.03%	4.029	0.34%	19.46	0.05%	67.55
128,000	0.04%	5.633	0.35%	28.62	0.06%	80.67
160,000	0.04%	7.323	0.35%	36.91	0.06%	95.21

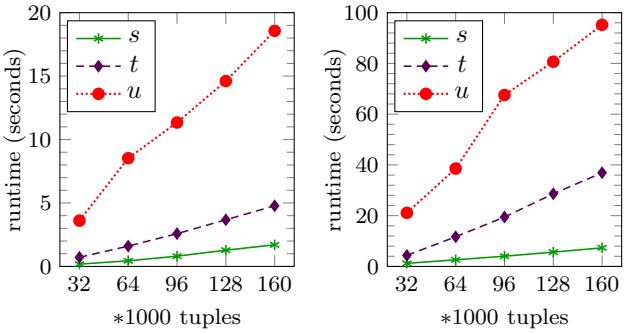


Fig. 11 Runtimes related to a growing number of tuples: new (left) and previous (right) version of the framework

9 Conclusions and future work

In this paper, we have introduced an efficient framework that supports pattern matching on datasets having any number of movement-related attributes of different types. The proposed approach constitutes as a major extension of the previous version of the framework [47], which has been enhanced with respect to several aspects. More precisely, details of the pattern language and the matching semantics have been revised, and efficiency gains have been achieved by changes in data structures and algorithms. According to the experimental evaluation based on two different datasets, the approach presented in this paper outperforms its precursor by approximately one order of magnitude.

As stated before, to our knowledge there is no comparable approach entailing a flexible and expressive pattern language that is useful for such a variety of application domains. The proposed framework is fully implemented in the open source DBMS SECONDO.

As a subject of future research, we plan to analyze whether a combination of symbolic trajectories [25] corresponding to the movement of an entity (for example, cardinal directions, speed categories, and altitudes) can be applied to restore the geometric trajectory. Moreover, we will create different distance functions for symbolic trajectories and/or complete tuples of time-dependent values in order to find similarities and clusters inside collections of movement data.

References

1. Alvares, L.O., Bogorny, V., Kuijpers, B., de Macêdo, J.A.F., Moelans, B., Vaisman, A.: A model for enriching trajectories with semantic geographical information. In: ACM GIS, pp. 22:1–22:8 (2007)
2. Andrienko, G.L., Andrienko, N.V., Heurich, M.: An event-based conceptual model for context-aware movement analysis. Int. J. Geograph. Inf. Sci. **25**(9), 1347–1370 (2011)
3. Bogorny, V., Renso, C., de Aquino, A.R., de Lucca Siqueira, F., Alvares, L.O.: Constant—a conceptual data model for semantic trajectories of moving objects. Trans. GIS **18**(1), 66–88 (2014)
4. Bracciale, L., Bonola, M., Loreti, P., Bianchi, G., Amici, R., Rabuffi, A.: Crawdad dataset roma/taxi. http://crawdad.org/roma/taxi/20140717 (2014). Accessed 23 Oct 2018
5. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2), 153–180 (2002)
6. Brinkhoff, T.: Network-based generator of moving objects. http://iapg.jade-hs.de/personen/brinkhoff/generator (2002). Accessed 23 Oct 2018
7. Cai, G., Lee, K., Lee, I.: Discovering common semantic trajectories from geo-tagged social media. In: IEA/AIE, pp. 320–332 (2016)
8. Camossi, E., Villa, P., Mazzola, L.: Semantic-based anomalous pattern discovery in moving object trajectories. CoRR arxiv:1305.1946 (2013)
9. Chang, J.W., Song, M.S., Um, J.H.: TMN-tree: new trajectory index structure for moving objects in spatial networks. In: CIT, pp. 1633–1638 (2010)
10. Damiani, M.L., Issa, H., Güting, R.H., Valdés, F.: Hybrid queries over symbolic and spatial trajectories: a usage scenario. In: MDM, pp. 341–344 (2014)
11. Damiani, M.L., Issa, H., Güting, R.H., Valdés, F.: Symbolic trajectories and application challenges. SIGSPATIAL Spec. **7**(1), 51–58 (2015)
12. Database Systems for New Applications, Fernuniversität Hagen. http://dna.fernuni-hagen.de/Secondo.html. Accessed 23 Oct 2018
13. de Almeida, V.T., Güting, R.H., Behr, T.: Querying moving objects in Secondo. In: MDM, pp. 47–51 (2006)
14. du Mouza, C., Rigaux, P.: Multi-scale classification of moving objects trajectories. In: SSDBM, pp. 307–316 (2004)
15. du Mouza, C., Rigaux, P.: Mobility patterns. GeoInformatica **9**(4), 297–319 (2005)
16. Erwig, M., Güting, R.H., Schneider, M., Vazirgiannis, M.: Spatio-temporal data types: an approach to modeling and querying moving objects in databases. GeoInformatica **3**(3), 269–296 (1999)
17. Fileto, R., May, C., Renso, C., Pelekis, N., Klein, D., Theodoridis, Y.: The baquara^2 knowledge-based framework for semantic enrichment and analysis of movement data. Data Knowl. Eng. **98**, 104–122 (2015)
18. Forlizzi, L., Güting, R.H., Nardelli, E., Schneider, M.: A data model and data structures for moving objects databases. In: ACM SIGMOD, pp. 319–330 (2000)
19. Geofabrik GmbH and OpenStreetMap Contributors: Openstreetmap data extracts. http://download.geofabrik.de (2007). Accessed 23 Oct 2018
20. Gryllakis, F., Pelekis, N., Doulkeridis, C., Sideridis, S., Theodoridis, Y.: Searching for spatio-temporal-keyword patterns in semantic trajectories. In: Advances in Intelligent Data Analysis, pp. 112–124 (2017)
21. Gryllakis, F., Pelekis, N., Doulkeridis, C., Sideridis, S., Theodoridis, Y.: Spatio-temporal-keyword pattern queries over semantic trajectories with hermes@neo4j. In: EDBT, pp. 678–681 (2018)
22. Güting, R.H., Behr, T., Düntgen, C.: Secondo: a platform for moving objects database research and for publishing and integrating research implementations. IEEE Data Eng. Bull. **33**(2), 56–63 (2010)
23. Güting, R.H., Böhlen, M.H., Erwig, M., Jensen, C.S., Lorentzos, N.A., Schneider, M., Vazirgiannis, M.: A foundation for representing and querying moving objects. ACM TODS **25**(1), 1–42 (2000)
24. Güting, R.H., Schneider, M.: Moving Objects Databases. Morgan Kaufmann, Los Altos (2005)
25. Güting, R.H., Valdés, F., Damiani, M.L.: Symbolic trajectories. ACM TSAS **1**(2), 7:1–7:51 (2015)
26. Hadjieleftheriou, M., Kollios, G., Bakalov, P., Tsotras, V.J.: Complex spatio-temporal pattern queries. In: PVLDB, pp. 877–888 (2005)
27. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley-Longman Publishing, Reading (2001)
28. Issa, H., Damiani, M.L.: Efficient access to temporally overlaying spatial and textual trajectories. In: MDM, pp. 262–271 (2016)
29. Liu, H., Xu, J., Zheng, K., Liu, C., Du, L., Wu, X.: Semantic-aware query processing for activity trajectories. In: International Conference on Web Search and Data Mining, WSDM, pp. 283–292 (2017)
30. NASA, NGA: Shuttle radar topography mission. https://lta.cr.usgs.gov/SRTM1Arc (2000). Accessed 23 Oct 2018
31. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings—Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, Cambridge (2002)
32. Newson, P., Krumm, J.: Hidden markov map matching through noise and sparseness. In: ACM SIGSPATIAL, pp. 336–343. ACM (2009)
33. Nguyen-Dinh, L., Aref, W.G., Mokbel, M.F.: Spatio-temporal access methods: part 2 (2003–2010). IEEE Data Eng. Bull. **33**(2), 46–55 (2010)
34. Nogueira, T.P., Braga, R.B., de Oliveira, C.T., Martin, H.: Framestep: a framework for annotating semantic trajectories based on episodes. Expert Syst. Appl. **92**, 533–545 (2018)
35. Openclipart: https://openclipart.org/ (2018). Accessed 23 Oct 2018
36. OpenStreetMap Foundation: Openstreetmap. http://www.openstreetmap.org (2004). Accessed 23 Oct 2018
37. Parent, C., Spaccapietra, S., Renso, C., Andrienko, G.L., Andrienko, N.V., Bogorny, V., Damiani, M.L., Gkoulalas-Divanis, A., de Macêdo, J.A.F., Pelekis, N., Theodoridis, Y., Yan, Z.: Semantic trajectories modeling and analysis. ACM Comput. Surv. **45**(4), 42 (2013)
38. Pelekis, N., Frentzos, E., Giatrakos, N., Theodoridis, Y.: HERMES: a trajectory DB engine for mobility-centric applications. IJKBO **5**(2), 19–41 (2015)
39. Pelekis, N., Theodoridis, Y.: Mobility Data Management and Exploration. Springer, Berlin (2014)
40. Pfoser, D., Jensen, C.S., Theodoridis, Y.: Novel approaches in query processing for moving object trajectories. In: VLDB, pp. 395–406 (2000)
41. Quddus, M.A., Ochieng, W.Y., Noland, R.B.: Current map-matching algorithms for transport applications: state-of-the art and future research directions. Transp. Res. Part C Emerg. Technol. **15**(5), 312–328 (2007)
42. Sistemi Territoriali: Roma capitale, mappa dei municipi. http://www.datiopen.it/en/opendata/Municipi_di_Roma_Capitale (2012). Accessed 23 Oct 2018
43. Spaccapietra, S., Parent, C., Damiani, M.L., de Macêdo, J.A.F., Porto, F., Vangenot, C.: A conceptual view on trajectories. Data Knowl. Eng. **65**(1), 126–146 (2008)
44. Valdés, F., Damiani, M.L., Güting, R.H.: Symbolic trajectories in SECONDO: pattern matching and rewriting. DASFAA **2**, 450–453 (2013)

45. Valdés, F., Güting, R.H.: Index-supported pattern matching on symbolic trajectories. In: ACM SIGSPATIAL, pp. 53–62 (2014)
46. Valdés, F., Güting, R.H.: Efficient multi-attribute analysis for trajectories: a case study for aircraft. In: ACM SIGSPATIAL, pp. 88:1–88:4 (2017)
47. Valdés, F., Güting, R.H.: Index-supported pattern matching on tuples of time-dependent values. GeoInformatica **21**(3), 429–458 (2017)
48. Valdés, F., Güting, R.H., Ossi, F.: Efficient trajectory analysis for several time-dependent attributes: a case study for roe deer. In: MDM, pp. 337–340 (2016)
49. Vazirgiannis, M., Theodoridis, Y., Sellis, T.K.: Spatio-temporal composition and indexing for large multimedia applications. ACM Multimed. Syst. **6**(4), 284–298 (1998)
50. Vieira, M.R., Bakalov, P., Tsotras, V.J.: Querying trajectories using flexible patterns. In: EDBT, pp. 406–417 (2010)
51. Vieira, M.R., Bakalov, P., Tsotras, V.J.: Flextrack: a system for querying flexible patterns in trajectory databases. In: SSTD, pp. 475–480 (2011)
52. Vlachos, M., Gunopulos, D., Kollios, G.: Discovering similar multidimensional trajectories. In: ICDE, pp. 673–684 (2002)
53. Yan, Z., Chakraborty, D., Parent, C., Spaccapietra, S., Aberer, K.: Semantic trajectories: mobility data computation and annotation. ACM TIST **4**(3), 49 (2013)
54. Zhang, C., Han, J., Shou, L., Lu, J., La Porta, T.F.: Splitter: mining fine-grained sequential patterns in semantic trajectories. PVLDB **7**(9), 769–780 (2014)
55. Zheng, K., Shang, S., Yuan, N.J., Yang, Y.: Towards efficient search for activity trajectories. In: ICDE, pp. 230–241 (2013)
56. Zheng, K., Zheng, B., Xu, J., Liu, G., Liu, A., Li, Z.: Popularity-aware spatial keyword search on activity trajectories. World Wide Web **20**(4), 749–773 (2017)
57. Zheng, Y., Xie, X., Ma, W.: Geolife: a collaborative social networking service among user, location and trajectory. IEEE Data Eng. Bull. **33**(2), 32–39 (2010)
58. Zheng, Y., Zhou, X. (eds.): Computing with Spatial Trajectories. Springer, Berlin (2011)