Index-supported Pattern Matching on Symbolic Trajectories

Fabio Valdés
Database Systems for New Applications
FernUniversität in Hagen
58084 Hagen
fabio.valdes@fernuni-hagen.de

Ralf Hartmut Güting
Database Systems for New Applications
Fernuniversität in Hagen
58084 Hagen
rhg@fernuni-hagen.de

ABSTRACT

Recording mobility data with GPS-enabled devices, e.g., smart phones or vehicles, has become a common issue for private persons, companies, and institutions. Consequently, the requirements for managing these enormous datasets have increased drastically, so trajectory management has become an active research field. In order to avoid querying raw trajectories, which is neither convenient nor efficient, a symbolic representation of the geometric data has been introduced.

A comprehensive framework for describing and querying *symbolic trajectories* including an expressive pattern language as well as an efficient matching algorithm was presented lately. A symbolic trajectory, basically being a time-dependent symbolic value (e.g., a label), can contain names of traversed roads, a speed profile, transportation modes, behaviors of animals, or cells inside a cellular network. The quality and efficiency of transportation systems, targeted advertising, animal research, crime investigation, etc. may be improved by analyzing such data.

The main contribution of this paper is an improvement of our previous approach, featuring algorithms and data structures optimizing the matching of symbolic trajectories for any kind of pattern with the help of two indexes. More specifically, a trie is applied for the symbolic values (i.e., labels or places), while the time intervals are stored in a one-dimensional R-tree. Hence, we avoid the linear scan of every trajectory, being necessary without index support. As a result, the computation cost for the pattern matching is nearly independent from the trajectory size. Our work details the concept and the implementation of the new approach, followed by an experimental evaluation.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Pattern Matching; H.2.8 [**Database Applications**]: Spatial Databases and GIS; H.3.1 [**Content Analysis and Indexing**]: Indexing Methods

SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA

ACM 978-1-4503-3131-9/14/11...$15.00
http://dx.doi.org/10.1145/2666310.2666402

General Terms

Algorithms, Performance

Keywords

Symbolic Trajectories, Pattern Matching, Trie, R-tree

1. INTRODUCTION

Due to the abundant use of position recording devices such as smart phones, automotive navigation systems, or other GPS sensors, the amount of daily generated mobility data has become tremendous. Similarly, the requirements for storing, managing and querying the so-called trajectories have increased.

A trajectory represents the movement of an entity, e.g., a person, a vehicle, or an animal, during a certain period of time. It can be considered as a sequence of time-stamped geographic positions or, abstractly, as a continuous function from time into two-dimensional space, denoted as moving point [9]. In order to discover any kind of interesting phenomena from such movement data, it is suitable to associate meanings to trajectory parts, depending on the purpose of the analysis. For instance, the observation of animals may be more effective if the trajectories contain labels like "at home range" or "at feeding station" instead of raw geographic coordinates. In addition, such a semantically annotated trajectory usually is much shorter than its spatial equivalent. Besides research fields like animal behavior or healthcare, the possible scope of application covers commercial purposes (e.g., logistical optimization, customer behavior, targeted advertising) as well as urban planning, private use, or criminal investigation.

The research field of moving objects databases has been very active in the last 15 years [10, 29]. The first definition of a conceptual trajectory model, based on the key concepts of stop and move, is given in [20], followed by generalized variants [1, 27, 17] of this model. However, since they are focused on the conceptual level, issues of data management remain unsolved. In [11], a comprehensive framework for the generalized representation of movement in a symbolic space is introduced. It is integrated into the data model of [9], available in moving objects database systems such as Secondo [8, 3] or Hermes [18]. It is also embedded into the Secondo implementation of the model of [9]. The approach comprises four data types for different kinds of symbolic trajectories and an expressive and fully implemented pattern matching language. A demonstration is conducted in [21].

Since a symbolic trajectory is related to a sequence of

strings, it seems natural to employ regular expressions for a pattern language and thus finite automata [13, 16] for pattern matching algorithms on symbolic trajectories. The authors of [5, 6] present a pattern language for trajectories defined in a discrete symbolic space and a pattern matching algorithm based on an NFA. They define an object's trajectory as a sequence of symbols denoting the successive zones visited by the object. However, their language does not allow conditions on variables or precise temporal specifications. In [23, 24, 25], an expressive pattern language for geometric trajectories including variables and conditions is presented. Its drawback is the limitation to symbolic trajectories containing names of areas inside a partitioned space. A contribution for detecting frequently occurring patterns from a set of semantic trajectories is given in [28], where similar places (regarding spatial, semantic, and temporal aspects) are grouped together.

In order to realize faster searches and pattern matching algorithms on trajectories, it is convenient to apply index structures. Several publications explore index structures for spatial trajectories, e.g., the 3DR-tree [22], the TB-tree [19], or the TMN-tree [2]. In [26], an index structure for discovering similar multidimensional trajectories is detailed. Our previous work [11] comprises an index for symbolic trajectories with restricted usability. To our knowledge, the unconstrained usage of a symbolic trajectory index has not been explored yet.

In this paper, we introduce a two-part index structure for collections of symbolic trajectories on the basis of an R-tree [12] and a trie (prefix tree) [4] as well as a suitable fully implemented pattern matching algorithm. The latter filters a collection of symbolic trajectories according to a specified pattern. In the first phase, we prune the trajectories that do not occur in the indexes for certain pattern elements' contents, before performing the exact matching on the reduced dataset. While applying the NFA transition function generated from the pattern, matching information as well as index retrievals are held and updated inside an efficient data structure. Everytime a trajectory is known to either match or mismatch the pattern, it is removed from the computation and, in the first case, inserted into the result set. Without index support, a trajectory in general has to be scanned completely for a matching decision. In the present approach, the computation cost is independent from the trajectory length, except for the number of index retrieval results that may increase with the number of symbolic values. With the help of the database benchmark BerlinMOD [7], we created a representative data set, on which we conducted a series of experiments.

The remainder of this paper is organized in the following way: After providing some preliminary information, i.e., above all, an overview of symbolic trajectories and our pattern language in Section 2, in Section 3 we introduce the index structures applied for the matching. Section 4 details the accelerated pattern matching approach. An experimental evaluation is conducted in Section 5, and Section 6 concludes the paper.

2. PRELIMINARIES

In this section, we first mention basic notations before reviewing some of the results of [11], focusing on the defined data types and the pattern language. In addition, we introduce an extension to the latter, making it more convenient to define relations between value sets in a pattern. Finally, a fairly complex pattern is detailed, and we discuss the meaning of an NFA with regard to the pattern language.

2.1 Basic Notations

We conceive a trajectory collection as a database relation with an attribute of one of the data types *mlabel*, *mlabels*, *mplace*, *mplaces*. Hence, symbolic trajectories can be stored along with their underlying geometric trajectories or any other information. Each of the relations' tuples has an id which is needed for indexing the contents of the symbolic trajectory. Since we later use a tuple id to fast access a certain vector slot, and since the tuple ids are in general not necessarily consecutive, we first map them onto a set $\{1.\ldots,n\}$ for the whole computation (the tuple id 0 does not exist; the value 0 is used otherwise). The result of the main algorithm is a vector of the tuple ids of the matching symbolic trajectories, thus we have to map that set back onto the original tuple ids. In the remainder of this paper, we refer to a tuple id as if they were in consecutive order, beginning with 1.

In contrast to the tuple ids, the components of a pattern P and of a symbolic trajectory M start at position 0 and are addressed by $P[i]$ and $M[j]$, respectively (precise definitions follow). The number of components of M is denoted by $|M|$, while a trajectory collection $\mathcal{M}$ contains $|\mathcal{M}|$ trajectories.

2.2 Symbolic Trajectories

This subsection, as well as the following one, largely refers to [11]. Basically, a *symbolic trajectory* is a function from time into a set of symbolic values, e.g., character strings. In this paper, we focus on objects of the type *mlabels*, representable as a sequence $\langle u_1, \ldots, u_n \rangle$, where each unit u_j is a pair (i_j, l_j) of a time interval and an arbitrarily large set of labels. The time intervals must be disjoint and ordered by time. The following three symbolic trajectories, containing labels that represent names of rivers, districts, institutions, cinemas, and places of interest inside Berlin, serve as a continuous example for the remainder of this paper.

```
M1 =  < ( [2014-05-02-10:25 2014-05-02-10:40)
          {"Havel", "Hbf"} ),
        ( [2014-05-02-10:40 2014-05-02-11:00)
          {"Havel", "Tegel"} ) >

M2 =  < ( [2014-05-03-15:30 2014-05-03-17:20)
          {"Alex", "BKA", "BMF"} ) >

M3 =  < ( [2014-05-17-20:00 2014-05-17-20:47)
          {"Havel"} ),
        ( [2014-05-17-21:02 2014-05-17-21:50)
          {"Havel", "Yorck"} ),
        ( [2014-05-17-21:50 2014-05-18-05:30)
          {"Yorck"} ) >
```

The brackets and parentheses around the time intervals represent closed and open interval limits, respectively, which is relevant since the intervals must not overlap but may be adjacent. Instead of labels, a unit can also comprise *places*. A place is a label combined with an integer (e.g., a reference to a spatial database object). Consequently, the data types *mplace* and *mplaces* are available.

2.3 The Pattern Language

From a symbolic trajectory M and a *pattern* P entered as a text, the corresponding pattern matching algorithm com-

putes a boolean value, which is true iff P matches M. We denote the simple example pattern

```
A [(2014-05 {"Havel", "Hbf"}) | (_ {"Hbf"})]+ B *
```

as P_0'. A pattern essentially is a regular expression over the set of *atomic pattern elements*, abbreviated as *atoms*. An atom of the form `(T L)`, where T is a set of time specifications and L is a set of labels, matches one trajectory unit (i_j, l_j) iff $i_j \subset t \; \forall \; t \in T$ and $l_j \subset L$. The sets T and L may also be replaced by an underscore, meaning that any time interval or label set of a unit (respectively) is matched. The wildcard atoms `+` and `*` match an arbitrary sequence of trajectory units, whose length has to be at least 1 for the former.

For instance, the unit $M_1[0]$ matches the atom $P_0'[0]$ = `(2014-05 {"Havel", "Hbf"})`, since its time interval and its labels are covered by the specifications. Neither $M_2[0]$ nor $M_3[0]$ match one of the atoms inside the regular expression. The final atom $P_0'[2] =$ `*` matches $M_1[1]$. Hence, P_0' matches only the trajectory M_1.

As an extension to the language's expressiveness, the use of conditions is enabled. That is, the user may assign the *pattern elements* (i.e., atoms outside of regular expression structures, or whole regular expressions) to variables and append any number of constraints, formulated as boolean expressions, that have to be fulfilled (e.g., `B.labels contains "Alex"` for P_0'). In case of a matching, each variable is assigned a sequence of trajectory units, and in each condition, any attribute of such a sequence (the label set, the start or end time, the number of units, etc.) can be compared with others. Also database objects and any operation available in the DBMS can be applied here. With the mentioned condition, P_0' does not match M_1, since the unit $M_1[1]$, which is bound to B, does not contain the label `"Alex"`.

2.4 Pattern Language Extension

In addition to the pattern language detailed above, in this paper we introduce the concept of different set relations. As described, for a successful matching of a trajectory unit u_j and an atom $P[a]$, the value set l_j inside u_j has to be a subset of the label set L provided in $P[a]$. In order to further increase the flexibility of the language and to make the specification of patterns more convenient, we added the following set relations:

set relation name	u_j matches $P[a]$ iff
`disjoint`	$l_j \cap L = \varnothing$
`superset`	$l_j \supset L$
`equal`	$l_j = L$
`intersect`	$l_j \cap L \neq \varnothing$

This extension can be applied by writing the name of the set relation directly in front of the desired set, for instance, `(_ equal{"Havel", "Hbf"})`, matching a trajectory unit with exactly the same values. Note that the notation without set relation keyword has the described meaning, i.e., u_j matches $P[a]$ iff $l_j \subset L$. All set relations are applicable to any kind of symbolic trajectory, for example, *mlabels*.

In the following, we denote the pattern

```
X * Y (2014-05 intersect{"Havel", "Tiergarten"})
Z [(2014-05-17-20:00~2014-05-17-23:55 "Yorck") |
   (_ superset{"Havel"}) (_ superset{"Tegel"})]+
// X.card = Z.card
```

as P_0. It contains five atoms $P_0[0], \ldots, P_0[4]$, where the first two are preceded by the variables X and Y, respectively. After $P_0[0] =$ `*` matches a sequence of arbitrarily many units of a trajectory, a unit occurring in May 2014, whose labels set intersects the set `{"Havel", "Tiergarten"}`, has to be matched. This unit is then bound to Y, while all its predecessors are associated to X. The third variable, Z, is assigned to a regular expression (denoted by `[...]+`) containing the atoms $P_0[2]$ to $P_0[4]$. Either the subsequent trajectory unit must have the label set `{"Yorck"}` (or no label at all) and occur on May 17, 2014, between 8 p.m. and 11:55 p.m, or (the symbol `|` separates alternatives) the next two units need to contain subsets of `{"Havel"}` and `{"Tegel"}`, respectively. Due to the symbol `+`, this alternative matching can be repeated any number of times. However, since P_0 ends here, the final repetition needs to match the last unit of the symbolic trajectory. Beyond that, according to the condition, the number of units bound to the variable X has to equal the number of units associated to Z.

2.5 NFA Creation and Meaning

Except for the conditions, a pattern is converted into an NFA transition relation δ. Each transition of such an NFA, i.e., an element $t \in \delta$, consists of a source state t_{source}, a trigger t_{atom} (corresponding to the position of an atom in the pattern) and a target state t_{target}. The pattern matching algorithm from [11] loops over the units of a trajectory, constantly updating the set of active NFA states. It reports a match iff a final state is active after the last unit. If conditions are used, the whole matching history is required to be kept for the condition evaluation in SECONDO.

The NFA created from P_0 is depicted in Figure 1. Note that the transitions represent the atom positions inside the pattern, e.g., $P_0[3]$ is abbreviated by 3. The NFA states are consecutively numbered and have no further meaning.

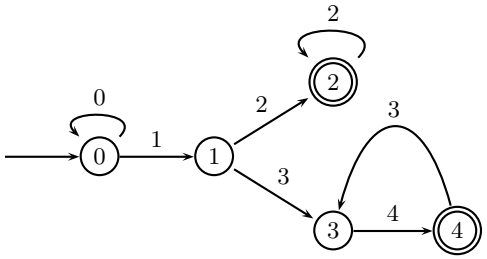

Figure 1: The NFA computed from P_0

3. APPLIED DATA STRUCTURES

In the first two subsections of this section, we detail the data structures used for the symbolic trajectory index, consisting of a structure for text values and another for time intervals. Subsequently, we present the efficient storage of index retrievals. The final subsection is dedicated to a structure that supports the exact matching process.

3.1 Symbolic Value Index

All symbolic values from a trajectory collection are stored into a single trie (prefix tree) [4] along with the desired position information. SECONDO provides a corresponding algebra including operators for inserting and retrieving values.

During the insertion process, each label (or place) of the trajectory collection is stored together with the tuple id of the trajectory and the position of the unit containing it. If

the trajectory type is *mplace* or *mplaces*, the place reference is memorized in addition. Since the value index construction requires a complete scan of every trajectory, this operation is expensive.

The retrieval of a text from a trie is highly efficient. When a text (in this context, either a label value or a place name) is successfully retrieved from the trie, an iterator is returned that loops over the result set, such that all occurrences of the value are found. In the negative case (and after passing the final search result), the iterator points to zero. If a value occurs in the index, the retrieval cost is linear in the length k of the entered text plus the number of query results. Otherwise, the search is aborted after less than k steps.

Figure 2 depicts the trie that represents the *mlabels* collection from our continuous example.

3.2 Time Interval Index

In addition to the symbolic value index, we apply an index for time intervals. For every trajectory from the collection, we consider the time intervals of all units. The applied data structure is a one-dimensional R-tree [12], where every interval is stored along with the corresponding tuple id and unit position. The related implementation is a component of SECONDO.

During the preprocessing and the exact matching, this index structure is applied to identify the trajectories and unit positions matching a time interval from an atom. The retrieval cost is logarithmic in the number of inserted intervals, being equivalent to the total number of units inside the trajectory collection. Hence, the time interval index for $\{M_1, M_2, M_3\}$ contains six intervals.

3.3 A Container Holding Index Results

Every result from one of the indexes (i.e., a tuple id and a unit position, hence, a pair of integers) is stored in a specialized structure, in order to avoid repeated scanning of irrelevant findings. For example, if a trajectory has already passed (or failed) the exact matching process, its tuple id may still be retrieved from the indexes. As both indexes always iterate over all retrievals, this would cause unnecessary computation cost.

Consequently, everytime an atom $P[a]$ is considered for the first time, we store the results for every trajectory that is active (what exactly this means is detailed later) in a slot of a vector R. Each slot of R contains a set $units$ of integers representing the unit numbers found in the indexes for this trajectory. In addition, there are two integers *pred* and *succ* pointing to the previous and next tuple id that is active and occurs in the index results for the contents of $P[a]$. Note that $R[0].succ$ indicates the first active position, since 0 is not a valid tuple id.

Since it is not exactly a matter of the pattern matching algorithm, we detail here how the slots of R are populated. Consider an atom of the form `(T rV)`, with a non-empty set T of time specifications, a non-empty set V of values, and a set relation r. First, both indexes are queried with all elements of T and V, respectively, each yielding a set of integer pairs (id, u) referencing a tuple id and a unit position (inactive tuple ids are neglected). Then the sets related to time intervals are intersected, and the set relation r is applied to the sets corresponding to elements of V. Hence, we obtain two sets of integer pairs, one satisfying the time specification and one fulfilling the value specification. The intersection of these sets is the set of references for `(T rV)`. Finally, we iterate over this result set and insert each unit position u into the set $R[id].units$. At the same time, the pointers *pred* and *succ* are defined as described before.

If either T contains no indexable element or V is empty, the related index does not need to be queried. Also the final intersection is skipped.

For every considered atom $P[a]$ (either in the preprocessing or in the exact matching), the vector R is computed once and stored into another vector $indexResult$ at the position $indexResult[a]$. This structure allows us to insert, retrieve, and deactivate contents in constant time. Moreover, an iteration over all components applying *pred* and *succ* comprises only the relevant ones. If a tuple id id is deactivated during the preprocessing or the exact matching process, we have to make sure that an iteration over a vector $indexResult[a]$ ignores id. Hence, for each position a, we set the reference *succ* of the predecessor of id to id's successor, and vice versa. In addition, the set of unit positions $indexResult[a][id].units$ is cleared.

3.4 A Structure for the Exact Matching

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

The set of IMI instances for one trajectory is stored in a vector slot whose number represents the trajectory's tuple id. Again, we apply two references *pred* and *succ* for a fast access to every active slot. Since each of these vectors of IMI sets depends on the current NFA state, they are held in the vector $indexMatching$, or iM, for short, where the position represents the state. Since the automaton is the basis of the computation of the exact matching, it is convenient to have a fast access to the IMI instances depending on the currently active state(s). Figure 3 depicts the structure iM for $\{M_1, M_2, M_3\}$ after the preprocessing, where M_2 is deactivated (details follow in Section 4).

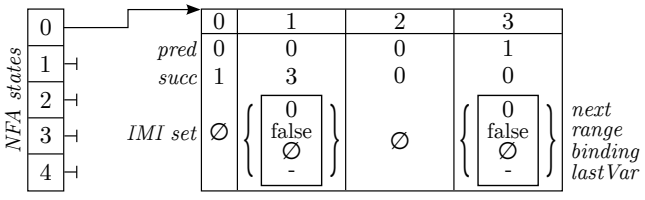

Figure 3: The Data Structure $indexMatching$

4. ALGORITHMS

In the following, we introduce the algorithms that are applied for index-supported pattern matching on a collection of symbolic trajectories. The matching process is divided

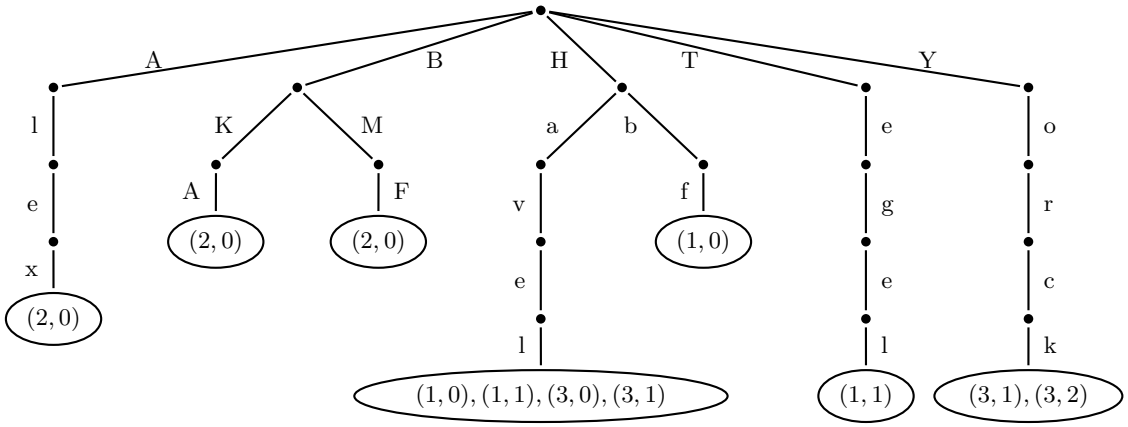

Figure 2: The trie corresponding to the collection $\{M_1, M_2, M_3\}$

into two main steps: First, we filter the trajectory collection and produce a set of candidates, and second, we perform the exact matching only on the candidates.

4.1 Preprocessing

The purpose of this efficient step is to reduce the number of trajectories that have to be processed subsequently. As detailed in the Sections 6.1 and 6.2.1 of [11], a pattern is converted into an NFA whose transition triggers represent atoms. In order to apply the index for a filtering step, we first identify the transitions that are mandatory for the automaton to reach any of the final states. We denote them as *crucial transitions* from now on. The contents of the corresponding atoms are retrieved from one (or both) of the indexes, and only the resulting tuple ids are passed to the exact matching step.

4.1.1 Identifying an NFA's Crucial Transitions

Before determining the crucial transitions of an NFA, we have to guarantee its number of possible paths to a final state to be finite, otherwise the subsequent algorithm would not necessarily terminate. Hence, we need to eliminate the symbols `+` and `*` from the pattern, since they are the cause for loops. If they represent wildcard pattern elements, they can be simply neglected since they carry no information relevant for an index operation. For a regular expression of the form `[...]+`, only the repetition symbol is deleted, since the term has to be traversed at least once for a matching. However, a regular expression followed by a `*`, not necessarily being traversed, is completely omitted. After these manipulations, a simplified NFA is computed from the modified pattern.

Regarding the pattern P_0, the abovementioned method results in the following pattern P_0' without loops:

```
Y (2014-05 intersect{"Havel", "Tiergarten"})
Z [(2014-05-17-20:00~2014-05-17-23:55 "Yorck") |
   (_ superset{"Havel"}) (_ superset{"Tegel"})]
```

The automaton resulting from this simplified pattern is shown in Figure 4. Note that the transition numbers remain unchanged compared to the complete NFA.

In the following phase, we detect every path through the simplified NFA leading to a final state. The number of such paths is finite due to the previous manipulation.

Algorithm 1 collects all paths leading from the initial state 0 to any of the final states. Each element of $Paths$ is a tuple consisting of a set p_{elems} and a state p_{state}. Every set p_{elems} represents a path through the NFA – more exactly, a sequence of transition triggers, which is equivalent to a sequence of atoms. The iteration continues as long as there are non-final states in $Paths$, in other words, it finishes when all paths have reached their destination. Note that without the second while loop condition, the loop would not be executed. Finally, the intersection of all paths p_{elems} results in a set containing exactly the transitions that were executed in each of the found paths. Hence, we computed the set of atoms that are necessary for a successful matching.

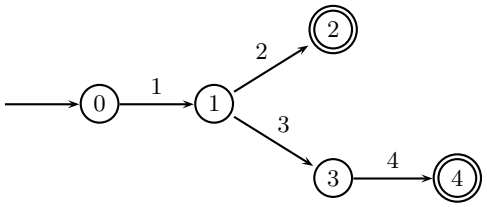

Figure 4: The Simplified NFA Computed from P_0'

Algorithm 1: *findCrucialElements*

Input: a loop free NFA δ and a set Γ of final states;
Output: a set of integers representing the crucial elements

```
1 let Paths = {(∅, 0)};
2 while {p_state | p ∈ Paths} ⊄ Γ ∨ Paths = {(∅, 0)} do
3     foreach p ∈ Paths do        // loop over current paths
4         foreach t ∈ δ(p_state) do  // loop over transitions
5             q ← p;
6             q_elems ← q_elems ∪ {t_atom};
7             q_state ← t_target;
8             Paths ← Paths ∪ {q};
9 return ⋂_{p∈Paths} p_elems;
```

We now apply Algorithm 1 to the automaton created from the simplified pattern P_0'. Initially, only state 0 is active, and there is only one transition leading to state 1, which is triggered by the atom $P_0[0]$. Hence, after the first iteration of the while loop, the set $Paths$ equals $\{(\{1\}, 1)\}$.

From state 1, there are two possible transitions, resulting in $Paths = \{(\{1,2\},2),(\{1,3\},3)\}$ after the second while loop. Subsequently, $Paths$ is updated to $\{(\{1,2\},2),(\{1,3,4\},4)\}$, so the while loop terminates, since 2 as well as 4 are final states. The returned set is the intersection of $\{1,2\}$ and $\{1,3,4\}$, i.e., $\{1\}$. This means, the atom $P_0[1]$ is the only one that has to be matched by any trajectory to enable a complete match.

If $P[a']$ is non-crucial, i.e., if there is a path through the NFA not including $P[a']$, we must not use it to filter the trajectory collection. No trajectory may be deactivated due to index results for $P[a']$, since it could match along another path that does not contain $P[a']$.

4.1.2 Filtering the Trajectory Collection

Subsequently, we iterate over the set of crucial elements and apply the indexes wherever possible. That is, for each non-empty crucial element $P[a]$ with indexable contents, the vector $indexResult[a]$ is filled according to Section 3.3. At the same time, we update a boolean vector A indicating which tuple ids are still present. For example, if a tuple id has index results for $P[a]$ but not for $P[a']$, it is removed from earlier results and, with the help of A, excluded from further consideration. As no exact matching was performed, false positives cannot be avoided in this phase. However, this method ensures that we process only trajectories whose tuple id occurs in the index results for the crucial pattern elements, so it remarkably reduces the dataset.

Regarding our continuous example pattern P_0, we identified $P_0[1]$ to be the only crucial atom. Therefore, we first retrieve the two labels `"Havel"` and `"Tiergarten"` from the trie. The label `"Havel"` yields the tuple ids 1 and 3. The search for `"Tiergarten"` is canceled without result after reading the character `"i"`, since after the first character `"T"`, there is only a branch for `"e"`. Due to the set relation `intersect`, at least one of the labels has to match at least one of the labels from a trajectory unit. Hence, the tuple ids resulting from the symbolic value index are computed by $\varnothing \cup \{1,3\} = \{1,3\}$.

In a second filter step, the time intervals of the crucial atoms are looked up in the interval tree. Concerning P_0, we search for the trajectory units (including tuple ids) whose time interval lies in or overlaps the period of May 2014. The R-tree yields all existing units from the trajectory collection, so there is no further filtering, and the tuple ids 1 and 3 are the result of the preprocessing.

The vector $indexResults[1]$ containing the index search results induced by the atom $P_0[1]$ has the following form:

slot / tuple id	0	1	2	3
units	$\varnothing$	$\{0,1\}$	$\varnothing$	$\{0,1\}$
pred	0	0	-	1
succ	1	3	-	0

4.2 Exact Matching

As stated before, our main challenge consisted in extending the index-supported pattern matching with simplified patterns (no regular expressions, no conditions) as mentioned in Section 6.2.8 of [11], to a general approach that processes all kinds of patterns. The use of regular expressions increases the complexity of the computation, since the number of paths traversing the NFA is not limited to 1 but can be infinite. Moreover, if the pattern contains conditions, further information have to be stored, and SECONDO must be used to evaluate the corresponding queries.

The matching of a pattern P with a trajectory collection $\mathcal{M}$ requires a parallel traversal of a path from the start state to a final state of the NFA for P as well as the sequence of units of every $M \in \mathcal{M}$. Hence, during the process, we always hold a set of active NFA states and have read (conceptually, not physically) an initial subsequence U of the units of each M. Such a state of scanning M is exactly represented by an IMI instance, which expresses U in terms of the next unit to be read. In case of a preceding wildcard transition in the NFA, the next unit is only a lower bound ($range$ = true in this case). For each M, several ways of matching P may exist. Therefore, for each trajectory at a given NFA state, multiple IMI objects must be maintained, each representing one possible matching. Initially, the only NFA state is 0, and for every M (if active after the filtering phase), the subsequence U is empty, so the next unit to be read is 0.

In the following, we present the algorithms for the matching process in execution order, as far as possible, before they are applied to the continuous example.

4.2.1 Initialization

For every tuple id id that passes the filtering detailed in Section 4.1.2, an instance of the class IMI is created and inserted into the vector slot $iM[0][id]$. Note that the first position indicator represents the NFA state, and initially, the only active state is 0. The IMI attributes are initialized as follows: $next$ is set to 0, $range$ equals false, and the other two variables remain empty. At the same time, an integer $activeTuples$, holding the number of currently active trajectories that equals 0 initially, is incremented. If a trajectory matches the pattern, its tuple id is inserted into the set $success$ of integers, which is empty in the beginning.

In terms of the trajectory collection from our continuous example, two IMI instances are created and inserted into the slots $iM[0][1]$ and $iM[0][3]$, respectively, see Figure 3. The value $activeTuples$ is set to 2.

4.2.2 Applying the NFA

The subsequent algorithm performs a loop over the NFA, as long as there are still trajectories that have not been completed. In contrast to our previous approach, Algorithm 2 does not process trajectories in a linear way. Instead, it iterates over the automaton and applies its transitions if the index results fit the previously saved matching information from iM'. At the same time, the new IMI data is stored into iM, the result set $success$ is extended if a match occurs, and $activeTuples$ is decremented after every matching decision.

For each currently active state $s \in States$, the algorithm tries to apply every possible transition that originates from s. More precisely, such a transition $t \in \delta(s)$ can be executed iff the corresponding atom t_{atom} matches at least one of the IMI instances of $iM'[t_{source}]$. This is verified in the procedures invoked in lines 10 and 12, respectively, updating iM, $success$ and $activeTuples$. The set of active states is refreshed accordingly. As soon as $activeTuples$ equals 0, the algorithm stops. According to line 9, further processing depends on the contents of $P[t_{atom}]$.

The purpose of the data structure from Section 3.4 is to hold all necessary information for each of the possible matching paths. Specifically due to the use of regular expressions that allow steps backwards in the NFA, it is required to enable multiple IMI instances for every NFA state and every trajectory. The structure is crucial for the matching deci-

Algorithm 2: *applyNFA*

Input: a pattern P;
an NFA δ with a set Γ of final states;
an integer *activeTuples*;
iM, see Section 3.4;
Output: a set *success* of integers, initially empty;

```
1  States ← {0};
2  while activeTuples > 0 do
3      States' ← States;
4      States ← ∅;
5      iM' ← iM;
6      clear iM;
7      foreach s ∈ States' do    // loop over current states
8          foreach t ∈ δ(s) do    // loop over transitions
9              if P[t_atom] has indexable contents then
10                 if indexMatch(iM, iM'[t_source], ...)
                   then States ← States ∪ {t_target};
11             else
12                 if nonIndexMatch(iM, iM'[t_source], ...)
                   then States ← States ∪ {t_target};
13 return success;
```

sion: The trajectory M with tuple id id matches the pattern P while a final state s is active iff there exists an IMI object in the set $iM[s][id]$ which is *finished*, i.e., whose *range* value is true or where $next = |M|$ holds. Finally, we call an instance *exhausted* if $next = |M|$ is true during a non-final state, meaning that a match is not possible anymore. When the exact matching process deactivates a tuple id, we apply the same strategy as described at the end of Section 3.3. Just replace *indexResult* by iM, and clear the set of IMI instances instead of the unit positions. In the following, the algorithms *indexMatch* and *nonIndexMatch* are detailed, before we show that Algorithm 2 always terminates.

First, we turn towards the algorithm *nonIndexMatch* that is invoked in case the atom related to the currently considered transition is a wildcard (1), that is, `*` or `+`, or empty (2), i.e., `(_ _)`, or non-empty but containing no indexable data (3), for example, `({monday, june} _)`. For processing one of these elements, we have to iterate over all existing IMI instances stored in $iM'[t_{source}]$. Since (1) and (2) guarantee a match with any trajectory unit, every IMI instance has to be updated. In case of (3), a positive matching test of the atom against the respective trajectory unit is required for the update. More specifically, if *range* is false, *next* is incremented by 1 and *range* is set to true for (1) and remains false for (2) and (3). If the *range* flag is originally true, i.e., if any unit u having $u \geq next$ inside the trajectory could match $P[t_{atom}]$, we only increment *next* by 1 for case (1). Otherwise, for every unit $u \geq next$, a new IMI instance having *range* = false and $next = u + 1$ is created (for (3), this happens only if u matches $P[t_{atom}]$). The algorithm returns true iff at least one updated has been conducted.

On the other hand, *indexMatch* is invoked if Algorithm 2 processes an atom with index-relevant contents. If $P[t_{atom}]$ is considered for the first time, we have to retrieve and to store the index results for it in the structure *indexResult*, as detailed in Section 3.3. Afterwards, we iterate over the active slots (tuple ids) of $indexResult[t_{source}]$, finding a set of unit positions u in each slot id. For each u, it is checked whether a matching IMI instance exists in $iM'[t_{source}][id]$. That is, if *range* is false, *next* needs to equal u. Otherwise, it suffices if $next \leq u$ holds. In both matching cases, a new IMI object with *range* = false and $next = u+1$ is created. A further check is required if $P[t_{atom}]$ includes non-indexable time specifications. If there is a mismatch with the unit at position u, no IMI instance is generated.

After any update or creation of an IMI object, it is mandatory to determine whether it is finished or exhausted. The former means that the trajectory with tuple id id matches the pattern, so we deactivate id (see Sections 3.3, 3.4), insert id into the set *success*, and ignore the finished IMI instance. Beyond that, *activeTuples* is decremented by 1. An exhausted IMI object is also neglected. If none of the constraints holds, it is inserted into the set $iM[t_{target}][id]$.

Hence, we deduce that Algorithm 2 terminates after no more than $\max\{|M| \mid M \in \mathcal{M}\}$ iterations of the while loop. However, this bound only has a theoretical meaning.

For the sake of efficiency, iterations over the set of active states as well as over the respective transition set are conducted in reverse order, i.e., we start with the highest active state (l. 7) and with the transition having the highest t_{atom} value (l. 8). Causing an earlier decrementation of the value *activeTuples*, this technique increases the probability of a faster matching decision.

4.2.3 Condition Processing

If the pattern includes conditions, further information has to be held in each instance of the class IMI. The attribute *binding* is necessary for an evaluation of the conditions by SECONDO (see Section 6.2.3 of [11] for details), while $lastVar$ is used to keep track of the binding variable that was changed most recently, in order to extend the binding correctly.

When an IMI instance is created or updated (except for the initialization), the attribute *binding* from the source instance is extended, depending on the variable *var* of the current atom and $lastVar$, the previously updated variable. We consider two (non-exclusive) cases: (1) For equal and non-empty variables *var* and $lastVar$, the binding of $lastVar$ is extended until $next - 1$, which is the last unit that was matched. (2) Now assume $var \neq lastVar$. Unless $lastVar$ is empty, we extend the binding of $lastVar$ to $next - 2$ (one unit before the most recently matched one). Moreover, if *var* exists (meaning no contradiction to the previous case), the mapping position $binding(var)$ is assigned the pair $(next - 1, next - 1)$. In any case, the attribute $lastVar$ in the new instance is assigned the current variable *var*.

If the IMI object is finished and belongs to a final state (i.e., a match in the condition-free version), the current binding is passed to the condition evaluation, where the second value of $binding(lastVar)$ is extended to $|M| - 1$ if *range* is true, i.e., if the matching was completed with a wildcard element. If the binding fulfills every condition, we treat the IMI instance and the tuple id like in the condition-free case. Otherwise, the IMI object is not processed anymore.

4.2.4 Working with the Continuous Example

In Table 1, we show the application of the presented technique to the continuous example, i.e., to the situation at the end of Section 4.2.1, using the following notation: The values true and false are abbreviated by t and f, respectively. A vector $indexResult[e]$ is denoted as a sequence of a tuple id and a set of units. An IMI instance is expressed in tuple

Table 1: Applying Algorithm 2 to the Continuous Example

After initialization: $States = \{0\}$, iM active at $\{1,3\}$, $success = \varnothing$, $activeTuples = 2$

Transition	$indexResult[t_{atom}]$; id: set of units	Tuple Id	No. of Units	Source IMI Instance	New IMI Instance	Abbrev. / Explanation
$0 \xrightarrow{1} 1$	$1:\{0,1\}$, $3:\{0,1\}$	1	2	$(0, f, \varnothing, -)$	$(1, f, \{Y \mapsto [0,0]\}, Y)$	(1)
		3	3	$(0, f, \varnothing, -)$	$(1, f, \{Y \mapsto [0,0]\}, Y)$	(2)
$0 \xrightarrow{0} 0$	none	1	2	$(0, f, \varnothing, -)$	$(1, t, \{X \mapsto [0,0]\}, X)$	(3)
		3	3	$(0, f, \varnothing, -)$	$(1, t, \{X \mapsto [0,0]\}, X)$	(4)
After first iteration: $States = \{0,1\}$, iM active at $\{1,3\}$, $success = \varnothing$, $activeTuples = 2$						
$1 \xrightarrow{3} 3$	$1:\{0,1\}$, $3:\{0,1\}$	1	2	(1)	$(2, f, \{Y \mapsto [0,0], Z \mapsto [1,1]\}, Z)$	exhausted
		3	3	(2)	$(2, f, \{Y \mapsto [0,0], Z \mapsto [1,1]\}, Z)$	(5)
$1 \xrightarrow{2} \mathbf{2}$	$3:\{2\}$	1	2	(1)	none	mismatch
		3	3	(2)	none	mismatch
$0 \xrightarrow{1} 1$	$1:\{0,1\}$, $3:\{0,1\}$	1	2	(3)	$(2, f, \{X \mapsto [0,0], Y \mapsto [1,1]\}, Y)$	exhausted
		3	3	(4)	$(2, f, \{X \mapsto [0,0], Y \mapsto [1,1]\}, Y)$	(6)
$0 \xrightarrow{0} 0$	none	1	2	(3)	$(2, t, \{X \mapsto [0,1]\}, X)$	exhausted
	none	3	3	(4)	$(2, t, \{X \mapsto [0,1]\}, X)$	(7)
After second iteration: $States = \{0,1,3\}$, iM active at $\{3\}$, $success = \varnothing$, $activeTuples = 1$						
$3 \xrightarrow{4} \mathbf{4}$	none (id 1 deactivated)					
$1 \xrightarrow{2} \mathbf{2}$	$3:\{2\}$	3	3	(6)	$(3, f, \{X \mapsto [0,0], Y \mapsto [1,1], Z \mapsto [2,2]\}, Z)$	match
After third and final iteration: $States = \{\mathbf{2}\}$, iM completely inactive, $success = \{3\}$, $activeTuples = 0$						

form. If there is a match, the new or updated IMI object either is assigned a number in parentheses, or it is dropped due to exhaustion or a complete match.

In the first line of Table 1, due to the transition, the considered atom is $P_0[1]$. The index(es) yield the tuple ids 1 and 3 with the units 0 and 1 for each of them. Regarding id 1, in $iM'[0][1]$ we find the initially created IMI instance $(0, f, \varnothing, -)$, matching the unit 0. Hence, $next$ is set to 1, the binding contains the assignment of Y to the interval $[0,0]$, and $lastVar$ now equals Y. We insert this new IMI instance into $iM[1][1]$, for the target state 1 and the tuple id 1. The tuple id 3 is processed similarly. At the bottom line of Table 1, we have $next \geq size$ for tuple id 3, meaning that the IMI instance is finished, and state 2 is final (bold font). For a pattern with conditions, the current binding has to fulfill them for a complete match. Each of the variables X and Z is bound to one trajectory unit, hence both cardinalities amount to 1, and the equation is fulfilled.

5. EXPERIMENTAL EVALUATION

In the following, we present the evaluation of our proposed model by means of a dataset based upon the BerlinMOD [7] data generator and a variety of patterns. All experiments were carried out on an AMD Phenom II X6 3.3 GHz processor with 8 GBytes of main memory, running openSUSE 12.3. From this environment, SECONDO was assigned one processor core and half of the available memory.

5.1 Dataset Generation and Properties

Our goal was to create a dataset being representatively large as well as comprehensible, in order to provide a comprehensive evaluation of our work. Hence, we chose to apply the BerlinMOD generator with a scale factor of 1.0, yielding 145,000 non-stationary raw trajectories (*mpoint* objects in SECONDO). Each of them was transformed into a symbolic trajectory of type *mlabels* similarly to Section 7.2.2 of [11], with the help of Parallel SECONDO [14, 15].

In the mentioned approach, each unit u of a trajectory had been labeled with the name of the street closest to the raw unit's centroid c_u. Now, for each c_u, beyond the name of the closest street, we computed the nearest river, restaurant, cinema, bar, natural area, body of water, place of interest, metro station, and train station along with their distances to c_u. Each name with a distance of less than 1 km was added to the symbolic unit. If the sets of names were equal for a sequence of units, the existing symbolic unit's time interval was extended instead of adding new units. All the applied categories are included in BerlinMOD as separate relations, containing (among others) the name and the geometry for each object (type *region* or *line*). Employing an R-tree for each category, this process was fulfilled in roughly 11 hours on a 12 disk cluster, resulting in 145,000 symbolic trajectories containing 13 million units with 89 million labels in total. The number of units per trajectory ranges from 1 to 400, with an average of almost 100. Although the symbolic trajectories contain strings with additional information from numerous relations, they occupy 2.6 GBytes of disk space, compared to 10.8 GBytes for the original raw trajectories.

5.2 Evaluation Results

The experiments focus on the effects of a growing number of queried trajectories as well as of an increasing trajectory size. Moreover, we apply patterns with different levels of complexity and selectivity. The precise query runtimes are displayed in Table 2, while the patterns are listed below:

$P_1 =$
```
(2007-06-11~2007-06-13 superset{"Kolk"}) *
```

$P_2 =$
```
A + B (~2007-06-20 intersect{"Sonnenallee"})
C + D (_ intersect{"Neukölln"}) E + //
get_duration(A.time) <
get_duration(C.time) + get_duration(E.time)
```

$P_3 =$
```
[(2007-06-01~2007-06-12 intersect{"BKA",
"Liebesinsel", "Tegel"}) |
(2007-06-02~ superset{"Kottbusser Tor"})] *
```

Table 2: Precise Runtimes and Selectivities of the Experiments

Growing Trajectory Collections							Different Trajectory Sizes						
	Runtime (sec.) / Selectivity							Runtime (sec.) / Selectivity					
$\|\mathcal{M}\|$	P_1		P_2		P_3		avg. $\|M\|$	P_1		P_2		P_3	
0	0.003	–	0.004	–	0.003	–	1	0.013	0	0.024	0	0.075	0
10,000	0.023	0.1%	0.163	0.47%	0.215	1%	50	0.02	0.03%	0.478	1.34%	0.157	1.1%
20,000	0.044	0.11%	0.402	0.58%	0.412	0.99%	100	0.027	0.1%	0.178	0.14%	0.263	1.2%
30,000	0.062	0.1%	0.553	0.57%	0.606	0.95%	150	0.026	0.11%	0.137	0.17%	0.286	0.86%
40,000	0.08	0.08%	0.671	0.54%	0.803	1.02%	200	0.045	0.32%	0.081	0.24%	0.233	0.76%
50,000	0.1	0.1%	0.833	0.56%	1.013	0.98%	250	0.034	0.07%	0.058	0.02%	0.341	0.98%

5.2.1 *Growing Trajectory Collection Sizes*

In order to generate a suitable data, six subrelations with cardinalities ranging from 0 to 50,000 are randomly taken from the main dataset. Hence, the average trajectory size per subset is nearly constant and equals almost 100. The trajectory collections are queried with patterns of different selectivities (0.1 % for P_1, 0.5 % for P_2, and 1 % for P_3). The corresponding results are depicted in Figure 5.

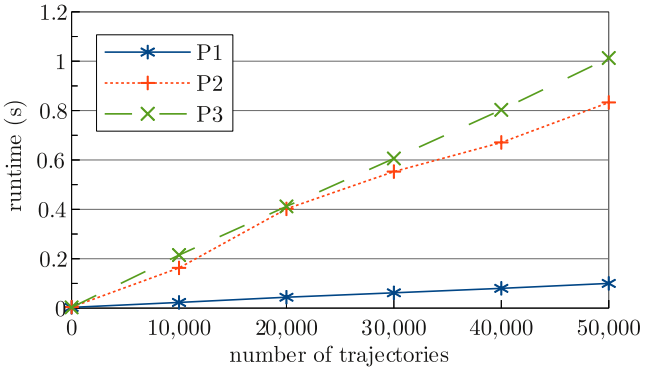

Figure 5: Runtimes caused by a growing number of trajectories

The graphs illustrate that the index-supported pattern matching is linear in the number of considered trajectories. Certain patterns cause sublinear runtime curves (as slightly hinted by the graph of P_2). However, due to the initialization of the vectors *indexResult* and *iM* as well as further auxiliary vectors for the retrieval of index results, the linear computation cost is inevitable. As expected, a higher selectivity induces a steeper curve, since more trajectories remain active, thus more vector slots have to be considered. Note that the condition of P_2 requires SECONDO query executions and, most probably, additional matching attempts, thus also increases the slope.

5.2.2 *Different Trajectory Sizes*

For the subsequent test series, we created subrelations from the original dataset with equal cardinalities (10,000 trajectories each) but with different trajectory sizes. Figure 6 shows the computation time graphs for this experiment, applying the same patterns as in the previous series of experiments. Note that the number of units per trajectory, displayed on the abscissa, is not an exact value (excluding the value 1). The respective figure represents the average of the trajectory sizes, allowing a deviation of 10 percent.

As stated before, the maximal number of iterations of the

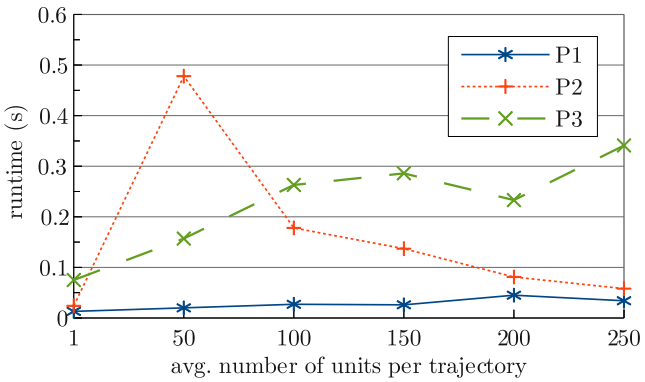

Figure 6: Runtimes depending on different trajectory sizes

main loop in Algorithm 2, which equals the size of the largest of the processed trajectories, is merely a theoretical limit. If a useful pattern is applied, the computation cost is independent from the number of units per trajectory, except for the effect of a higher number of index results that have to be considered. Since the queried trajectory collections are completely different to those from Section 5.2.1, the applied patterns yield other selectivities (see Table 2 for details). Given a constant number of trajectories, the runtime does not depend on the trajectory size anymore, meaning a major improvement compared to the results of [11].

5.3 Advantage over Previous Approach

In order to measure the efficiency gains compared to the pattern matching approach without index support, all above-mentioned experiments were conducted with the operator **matches** from [11]. These runtimes were divided by the runtimes from Sections 5.2.1 and 5.2.2, respectively. As a result, the queries involving the pattern P_2 caused the highest difference (up to 700 times faster), since the wildcard at its beginning ensures a complete scan of every trajectory in the non-index-supported case. On the other hand, for P_3 the advantage is less impressive (down to 2) due to the high number of index retrievals for the specified labels.

When different numbers of trajectories are queried, as in Section 5.2.1, the performance gain is more or less constant (approx. 25 for P_1, 90 for P_2, and 4 for P_3). However, for the second series of experiments, the quotient increases with a higher number of units per trajectory (11 to 33 for P_1, 16 to 700 for P_2, and 2 to 3 for P_3). This reflects the fact that the performance bottleneck caused by the linear trajectory scan has been removed.

6. CONCLUSIONS

In [11], a comprehensive framework for a general representation of movement in a symbolic space was introduced. It included an expressive pattern language for symbolic trajectories and a pattern matching implementation. However, the use of an index structure was restricted to patterns without conditions and regular expressions, and the index enabled solely the storage of labels, i.e., text items.

Beyond an extension of the pattern language, the contribution of this paper comprises the design and the implementation of a new pattern matching algorithm based on the use of two different index structures, one for symbolic values, i.e., labels or places, and one for time intervals. Our method avoids the expensive linear scan of the considered trajectories that was required before. Compared to our previous work, in most of the cases the computation cost has been reduced by more than an order of magnitude. The new approach is applicable to all kinds of patterns that are valid according to our pattern language. To our knowledge, there is no comparable work using a double index for pattern matching on symbolic trajectories.

Future work will focus on privacy issues related to symbolic trajectories. In this context, we will analyze whether and how the raw movement data can be computed from symbolic trajectories containing different types of information.

7. REFERENCES

[1] G. L. Andrienko, N. V. Andrienko, and M. Heurich. An event-based conceptual model for context-aware movement analysis. *International Journal of Geographical Information Science*, 25(9):1347–1370, 2011.

[2] J.-W. Chang, M.-S. Song, and J.-H. Um. Tmn-tree: New trajectory index structure for moving objects in spatial networks. In *CIT*, pages 1633–1638, 2010.

[3] V. T. de Almeida, R. H. Güting, and T. Behr. Querying moving objects in Secondo. In *MDM*, pages 47–51, 2006.

[4] R. De La Briandais. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 295–298, 1959.

[5] C. du Mouza and P. Rigaux. Multi-scale classification of moving objects trajectories. In *Proc. of the 16th Conference on SSDBM*, pages 307–316, 2004.

[6] C. du Mouza and P. Rigaux. Mobility patterns. *GeoInformatica*, 9(4):297–319, 2005.

[7] C. Düntgen, T. Behr, and R. H. Güting. BerlinMOD: a benchmark for moving object databases. *VLDB J.*, 18(6):1335–1368, 2009.

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

[9] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

[10] R. H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005.

[11] R. H. Güting, F. Valdés, and M. L. Damiani. Symbolic trajectories. *FernUniversität in Hagen, Technical Report 369, 2013*. `http://dna.fernuni-hagen.de/papers/SymbolicTrajectories369.pdf`.

[12] A. Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

[14] J. Lu and R. H. Güting. Parallel Secondo: Boosting database engines with hadoop. In *ICPADS*, pages 738–743, 2012.

[15] J. Lu and R. H. Güting. Parallel Secondo: Practical and efficient mobility data processing in the cloud. In *BigData Conference*, pages 17–25, 2013.

[16] G. Navarro and M. Raffinot. *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[17] C. Parent, S. Spaccapietra, C. Renso, G. L. Andrienko, N. V. Andrienko, V. Bogorny, M. L. Damiani, A. Gkoulalas-Divanis, J. A. F. de Macêdo, N. Pelekis, Y. Theodoridis, and Z. Yan. Semantic trajectories modeling and analysis. *ACM Comput. Surv.*, 45(4):42, 2013.

[18] N. Pelekis and Y. Theodoridis. *Mobility Data Management and Exploration*. Springer, 2014.

[19] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.

[20] S. Spaccapietra, C. Parent, M. L. Damiani, J. A. F. de Macêdo, F. Porto, and C. Vangenot. A conceptual view on trajectories. *Data Knowl. Eng.*, 65(1):126–146, 2008.

[21] F. Valdés, M. L. Damiani, and R. H. Güting. Symbolic trajectories in Secondo: Pattern matching and rewriting. In *DASFAA (2)*, pages 450–453, 2013.

[22] M. Vazirgiannis, Y. Theodoridis, and T. K. Sellis. Spatio-temporal composition and indexing for large multimedia applications. *Multimedia Syst.*, 6(4):284–298, 1998.

[23] M. R. Vieira, P. Bakalov, and V. J. Tsotras. Querying trajectories using flexible patterns. In *Proc. of the Conference on EDBT*, pages 406–417, 2010.

[24] M. R. Vieira, P. Bakalov, and V. J. Tsotras. Flextrack: A system for querying flexible patterns in trajectory databases. In *12th International Symposium, SSTD*, pages 475–480, 2011.

[25] M. R. Vieira and V. J. Tsotras. Complex motion pattern queries for trajectories. In *ICDE Workshops*, pages 280–283, Hannover, Germany, 2011.

[26] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

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

[28] C. Zhang, J. Han, L. Shou, J. Lu, and T. F. L. Porta. Splitter: Mining fine-grained sequential patterns in semantic trajectories. *PVLDB*, 7(9):769–780, 2014.

[29] Y. Zheng and X. Zhou, editors. *Computing with Spatial Trajectories*. Springer, 2011.