Mobility Patterns

Cédric du Mouza[*], Philippe Rigaux[†]

Abstract

We present a data model for tracking mobile objects and reporting the result of continuous queries. The model relies on a *discrete* view of the spatio-temporal space, where the 2D space and the time axis are respectively partitioned in a finite set of user-defined areas and in constant-size intervals. We define a query language to retrieve objects that match *mobility patterns* describing a sequence of moves and discuss evaluation techniques to maintain incrementally the result of queries.

1 Introduction

In the database community, several data models have been proposed to enable novel querying facilities over collections of moving objects. A common feature of most of these models is the strong focus on the *geometric* properties of trajectories. Indeed, in most cases, the data representation and the query language are considered as extensions of some existing data model previously designed for (and limited to) 2D geometric data handling. The modeling of moving objects has been therefore strongly influenced by the existing spatial models, and relies usually on a set of data structures providing support for geometric operations (e.g., geometric intersection) [21, 10, 8, 9].

An assumption commonly adopted by all the above mentioned models is to consider a *dense* embedding space and to model trajectories as *continuous* functions in this space. While this property allows several suitable computations (for instance the position of an object can be obtained at any instant), it is not well adapted to some common requests. Let us consider the tracking of objects with *continuous queries*, i.e., queries whose result must be maintained during a given (and possibly unbounded) period of time. When asking, for instance, for all the objects that belong to a given rectangle R during the next 3 days, the initial result is subject to vary by considering the objects that leave of enter R. Managing

[1]lab CEDRIC, CNAM, Paris, France, dumouza@cnam.fr
[2]LRI, Univ. Paris-Sud Orsay, France, rigaux@lri.fr

incrementally the evolutions of the result (i.e., without recomputing periodically the entire result) is a hard task with a geometric-based query language because the dense-space assumption of the data model often contradicts with the discrete nature of the observation. A trajectory for instance is obtained through sample points provided by the GPS system, and the continuous representation has to be inferred by interpolation between two sample points, or by extrapolation from the last known position [21]. Moreover, depending on the geometric operations required by the query, one might have to consult the past trajectory to check whether or not the object belongs to the result. Actually the few works that propose a solution to the problem deal with a limited class of queries (e.g., window queries in [13]).

In the present work we investigate an alternative approach, namely the management of continuous queries as a discrete process relying on *events* related to the moves of objects over the underlying space. Intuitive examples of events are, for instance, an object *enters* a zone, an object *stays* in a zone, and an object *leaves* a zone. A query in such a setting is a sequence of primitive events which can be specified either by explicitely referring to the zones of interest ("Give all the objects currently in a which arrived 5 minutes ago, coming from b"), or by more generic *patterns* of mobility such as, for instance, "Give these objects that moved from a to another zone and came back to a".

We propose in the current paper a data model for representing trajectories as sequences of moves in a discrete spatio-temporal space, and study the languages to query such sequences of events. Essentially, the languages that we consider allow to construct expressions, or *mobility patterns*, to express search operations. We focus specifically on the family of patterns that satisfy the following properties (i) we do not need the past moves of an object o to determine whether o matches or not a given pattern and (ii) the amount of memory required to maintain a query result is small. These properties are essential in the context of continuous queries since they guarantee that a large amount of queries can be evaluated efficiently with limited resources by just considering the last event associated to an object. We define a class of queries which provides an appropriate balance between expressiveness and the fulfillment of these requirements.

Related work

Expressing sequences of moves as proposed in the present paper is close in spirit to the area of sequence databases [20, 15, 18, 22]. The SQL-TS language of [18] and [19] allows to express sequences of conditions and describes an efficient algorithm for query evaluation. The idea of representing temporal sequences as strings and to rely on pattern-matching algorithms is also present in [6] and [5]. In [17] sequences

are considered as sorted relations, and each tuple gets a number that represents its position in the sequence. All these approaches are significantly different from ours. In particular there is nothing similar to the concept of mobility pattern, featuring variables, proposed in our data model.

The notion of continuous queries, described as queries that are issued once and run continuously, is first proposed in [24]. The approach considers append-only databases and relies on an incremental evaluation on delta relations. Availability of massive amounts of data on the Internet has considerably increased the interest in systems providing event notification across the network. Some representative works are the *Active Views* system [1], the NiagaraCQ system [4], and the prototypes described in [14, 7]. In the area of spatio-temporal databases, the problem is explicitely addressed in several works [16, 3, 13, 23, 11, 25]. [3] for instance describes a web-based architecture for reducing the volume and frequency of data transmissions between the client and the server. [13] presents a system that indexes queries in order to recompute periodically the whole result of each query. This is in contrast with the incremental computation advocated in the current paper.

In the rest of this paper we first develop an informal presentation of our work (Section 2) with examples of *mobility patterns* that illustrate the intuition behind the model and its practical interest from the user's point of view. The data model is presented in Section 3. Finally Section 4 concludes the paper and discusses future work. A long version is available at *http://www.lri.fr/~rigaux/DOC/MR04b.pdf*.

2 Introduction to mobility patterns

Figure 1 shows a map partitioned in several zones identified with simple labels (`a, b, c, ...`). Over this map we consider a set of mobile objects, each of them coupled with a localization device which periodically provides their position. The minimal period between two events related to the same object defines the *time unit*. For the sake of concreteness we shall assume in the following that objects are tracked by a GPS system giving the location of an object, and that the time unit is 1 (one) minute.

Consider now a traffic monitoring application supporting tracking of the mobile objects, and the following queries:

1. Give all the objects that traveled from `a` to `f`, stayed more than 10 minutes in `f` and then traveled from `f` to `c`.

2. Give all the objects traveling from `f` to `d` or `c` through another, third, zone of the map.

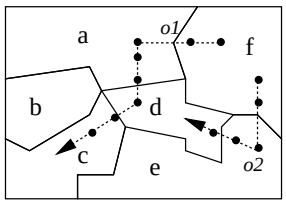

Figure 1: Objects moving over a partitioned map

3. Give all the objects that left a given zone, went to c and came back to the first zone.

The common feature of these examples is a specification of the successive zones an object belongs to during its travel, along with temporal constraints. We call *mobility pattern* this specification. The geometric-based approach used in most of the spatio-temporal data models so far is not really adapted for expressing queries based on mobility patterns. Actually we do no longer need an interpolation or extrapolation mechanism to infer the position of an object at each instant since the discrete succession of events provided by the GPS server is naturally suitable to serve as a support for evaluating these patterns.

Each GPS event provides the position of an object, and this suffices to compute the zone where the object resides when the event is received. It is therefore quite easy to construct a discrete representation of the trajectory of an object as a sequence of the form $l_1\{T_1\}.l_2\{T_2\}.\cdots.l_n\{T_n\}$ featuring the list $l_1, l_2, \ldots, l_n$ of successive zone labels as well as the time spent in each zone. For instance the trajectory of o_1 in Figure 1, assuming that o_1 spent 2 minutes in f, 4 minutes in a, 3 minutes in d and 6 minutes in c, will be represented in our model as a sequence [f{2}.a{4}.d{3}.c{6}]. Note that each new event either increments the time component of the last label if the object remains in the same zone, or appends a new label to the trajectory's representation.

Let us now turn to mobility patterns. Basically, they constitute a specific kind of regular expression, featuring variables which can be instantiated to any of the labels of the map. As a first example, assume that we want to retrieve all the objects that started from a or b, moved to e, crossing one of the zones c, d, or f (see Figure 1), and finally went back from e to a via the *same* zone. This class of trajectories is represented by the following mobility pattern:

$$(a|b).@x^{+}.e^{+}.@x^{+}.a$$

In a pattern, a zone is represented either by its label (here a, b, e) or by a variable (here @x) if it is left undetermined by the user. A variable is here necessary to

represent the zone where an object moved after leaving `a` or `b`, and expressing that the object must come back to `a` via the *same* zone. Each occurrence of a variable in a pattern must be instantiated to the same value. Labels or variables can be concatenated (for instance `@x.a` in our example) to describe a path, or grouped in sets (for instance `(a|b)`) to describe a union of zones. The "+" operator expresses the fact that the object can stay an undetermined time in a given zone, but at least one time unit. Alternatively, one can associate to each label simple temporal constraints of the form $\{$`min, max`$\}$ where `min` and `max` denote respectively the minimal and maximal number of time units spent in the zone.

Intuitively, an object o *matches* a pattern P if the following conditions hold:

1. one can find a word in the language $\mathcal{L}(P)$ which is equal to a suffix of the trajectory of o, *modulo* an assignment of the variables in P.
2. the time spent in each zone complies with the temporal constraint expressed in the pattern.

For instance an object whose trajectory is represented by the sequence `[f.d.c.b.`**`a.d.e.d.a`**`]` (we omit the temporal information for simplicity) matches the pattern above where the value of the variable `@x` is set to the label `d`. The suffix in boldface is then a word in the language denoted by the pattern.

The suffix represents here the most recent part of the trajectory received from the continuous stream of GPS events. It determines whether an object belongs or not to the result set of a query. Note also that, since the trajectory representation evolves as new events are received, the matching must be evaluated periodically – almost continuously. Our goal is to perform this evaluation with minimal space and time consumption.

Patterns can easily be introduced in a SQL-like query language, as illustrated by the following examples which will be used throughout the rest of the paper. The syntax of regular expression is that of the Perl language [26].

- **Q1.** Give all the objects that traveled from `a` to `f`, stayed at least 2 minutes in `f` and then traveled from `f` to `c`.

  ```
  SELECT *
  FROM Mob
  WHERE matches(traj,'a.f{2,}.c')
  ```

  The *matches* function checks whether a suffix of the spatio-temporal attribute *traj* matches the mobility pattern `a.f.c`. An additional temporal constraint states that the object must spend at least 2 time units (e.g., 2 minutes) in `f`.

- **Q2.** Give all the objects that stay in a or b all the time except for one minute when they were in another, third, zone.

```
SELECT *
FROM Mob
WHERE matches (traj,'(a|b)+.@x.(a|b)+')
AND    @x != 'a' AND @x != 'b'
```

  This example requires a variable @x which expresses a move not assigned to a specific label but instantiated to the choice of a moving object when it leaves a or b. It is possible to express additional constraints on the instantiations allowed for a variable, using equalities or inequalities. The user requires in this example the object to leave a or b for a third, distinct, area.

- **Q3.** Give all the objects that went through f to another zone then went to d or c, and came back to f using the *same* zone.

```
SELECT *
FROM   Mob
WHERE  matches (traj,'f.@x+.(d|c)+.@x+.f')
AND    @x != 'f'
```

Let us turn now to the query evaluation process, and in particular to the *continuous* evaluation which maintains a result by adding or removing objects. We consider two essential criteria for measuring the easiness and efficiency of this evaluation:

1. Do we need to consider the past moves of an object to evaluate a query?
2. What is the amount of memory required to maintain a query result?

Consider first the case of patterns without variable. Evaluating a pattern P is then a standard operation which simply requires to build the Finite State Automata (FA) that recognizes the language $\Sigma^*.L_P$, where L_P is the regular language denoted by P and Σ is the set of labels of the map.

In the general case, the FA associated to a regular expression is non-deterministic. Then an object o might be associated to several states at a given time instant, and we must record the list of current states for o. This list can be represented as a mask of bits, one bit for each state of the FA. The value 1 (resp. 0) for a bit means that o is (resp. is not) in the associated state. This gives a rather compact structure: for a pattern with 8 symbols, a mask of 8 bits (one byte) must be recorded for each object. One can track a database of one million objects with only one megabyte in main memory.

The pseudo-code of the procedure *HandleEvent(q, id, x, y)* summarizes how to actualize the result of a query q when a GPS event is received, giving a new location (x, y) for the object o. The reference map is a set of zones denoted by M.

HandleEvent (q, o, x, y)
begin
  // Compute the current zone, z
  z = *PointInPolygon(M, x, y)*
  // Get the label of z
  l = *label(z)*
  // For each bit set to 1 in the status of o,
  // compute the transition l
  for each bit i with value 1 in $status_o$
    Compute $s_j = \delta(FA_q, s_i, l)$
    Set the bit j to 1 and the bit i to 0 in $status_o$
  end for
end

The result set of q can then be updated according to the new status of object o. Essentially, if at least one of the new states is an accepting one, o will be *in* the result set, else it will be *out* of this result set. In this simple case we obtain a direct answer to the two questions above:

1. It is *not* required to maintain historical information on a trajectory, since, it suffices to know the current state(s) of the FA, reached by taking account of the events received so far.

2. The space required to maintain a query result is, in the worst case, the set of all states in the FA (which might be non-deterministic) and is therefore proportional to the size of the query[1].

If we consider now patterns with variables, the language is much more expressive, but some care is required for executing queries. Take for instance the example Q3 above. Each time an object leaves the zone `f` for another one, a new label is bound to the variable `@x`. One must then store this value in order to check for the consistency of any further occurrence of `@x`.

The next section is devoted to the data model, and focuses on the evaluation of queries with variables. We show that we can still avoid to rely on historical information on trajectories, and study more specifically the memory requirements for several classes of queries.

[1]It is possible, for any regular expression E, to construct a FA whose number of states is equal to the number of symbols in E.

3 The model

We consider an embedding space partitioned in a finite set of zones, each zone being uniquely labeled with a symbol from a finite alphabet Σ. The time axis is divided in constant size units. For concreteness we still assume in the following that the time unit is 1 minute. We also assume a set $\mathcal{V}$ of variables with $\Sigma \cap \mathcal{V} = \emptyset$ and denote as Γ the union $\Sigma \cup \mathcal{V}$. In the following, letters a, b, c, ... will denote symbols from Σ, and @x, @y, @z, ... variables. We assume the reader familiar with the basic notions of regular expressions and regular languages, as found in [12].

3.1 Data representation and query language

We adopt a standard extended relational framework for the database, with $\mathcal{O}$ denoting the relation of moving objects, and $o.traj$ the trajectory of an object o. The representation of trajectories is then defined as follows:

Definition 1 (Representation of trajectories) *A* trajectory *is represented by an expression of the form*

$$s_1\{T_1\}.s_2\{T_2\}.\cdots.s_n\{T_n\}$$

where $s_i, i = 1, \ldots, n$ are symbols from Σ and T_i represents the number of time units spent in the zone s_i.

Hereafter, we shall use the term "trajectory" to mean its representation. For convenience, we shall often omit the temporal components and use a simplified representation of a trajectory as a word $[s_1.s_2.\cdots.s_n]$ in Σ^*.

A natural choice is to build mobility patterns as regular expressions on $\Gamma = \Sigma \cup \mathcal{V}$, and to search for the suffix of trajectories that match the expression for some value of the variables. Consider for example the regular expression $E =$a.@x+.b+.@x. The trajectory $t =$f.d.a.c.b.c matches E because we can find a word $w =$ a.@x.b.@x in the language denoted by E (w is called a *witness* in the following) and an instantiation $\nu : \{$@x $:= c\}$ such that $\nu(w)$ is a suffix of t. However this approach raises some ambiguities regarding the role of variables. Consider the following examples:

1. Let E be the regular expression b.(a|@x)+.c. Then the trajectory b.a.c has two witnesses in the regular language denoted by E: b.@x.c and b.a.c. In the first case @x must be instantiated to a, but in the second case any value of @x is acceptable.

2. Let E be the regular expression $\texttt{a.(@x|@y).b.(@x|@y)}$. The variables $\texttt{@x}$ and $\texttt{@y}$ can be used interchangeably, which makes the role of variables undetermined.

As shown by the previous examples, if we build mobility patterns with unrestricted regular expressions over Γ, the assignment of variables is non deterministic, and sometimes meaningless. For safety reasons, when reading a word w and checking whether w matches a mobility pattern P, we require each variable in P to be explicitely bound to one of the symbols in w. We thus adopt a more rigorous definition of the language by considering only *unambiguous* regular expressions on Γ such that each variable always plays a role in the evaluation of the query. We need first to introduce *marked* regular expressions.

Definition 2 (Marked expressions [2]) *Let E be a regular expression over the alphabet Γ. We define the* marking *of E as the regular expression E' where each symbol of Γ is marked by a subscript over $\mathbb{N}$, representing the position of the symbol in the expression.*

For instance the marking of the regular expression $\texttt{a}^*\texttt{.@x.((b.a)|(c.b)).c.@x}^*\texttt{.a}$ is the expression $\texttt{a}_1^*\texttt{.x}_2\texttt{.((b}_3\texttt{.a}_4\texttt{)|(c}_5\texttt{.b}_6\texttt{)).c}_7\texttt{.@x}_8\texttt{.a}_9$. We can now define *mobility patterns* as the class of regular expressions that satisfy the following property:

Definition 3 (Mobility patterns) *A* mobility pattern *is a regular expression P over Γ such that each variable of P' appears in each word of the language $\mathcal{L}(P')$.*

This property ensures that each variable in any pattern is always assigned to a relevant label during query evaluation. The expression P $\texttt{= (a|b)+.@x.(a|b)+}$ is for instance a mobility pattern because $\texttt{@x}$ appears in all the words of the language $\mathcal{L}(P)$. Any successful matching of P with a trajectory $\texttt{t}$ results therefore in an assignment of $\texttt{@x}$ to one of the symbols of $\texttt{t}$. It can be tested whether a regular expression matches the required condition, and thus can be used as a mobility pattern.

Proposition 1 *There exists an algorithm to check whether a regular expression is a mobility pattern.*

Proof (sketch): Let E be a regular expression. Then $\mathcal{L}(E)$ and $\mathcal{L}(E')$ are regular languages. We define the language $\mathcal{L}_m = \{\Gamma^*.\texttt{@x}_1.\Gamma^*.\texttt{@x}_2..\texttt{@x}_k.\Gamma^*\}$, where Γ stands for $\Sigma \cup \mathcal{V}$, and $\texttt{@x}_1\texttt{,..,@x}_k$ are the variables of E'. $\mathcal{L}_m$ is regular by construction, so $\overline{\mathcal{L}_m}$ and $\mathcal{L}(E') \cap \overline{\mathcal{L}_m}$ are also regular. Consequently the fact that

$\mathcal{L}(E') \cap \overline{\mathcal{L}_m}$ is empty is decidable. And if $\mathcal{L}(E') \cap \overline{\mathcal{L}_m} = \emptyset$, then all the marked variables appear in all the words of $\mathcal{L}(E')$. □

Example 1 *The following regular expressions represent the mobility patterns of the sample queries* ***Q1****,* ***Q2*** *and* ***Q3*** *given in Section 2.*

1. $P_1 =$ `a.f{2,}.c`
2. $P_2 =$ `(a|b)+.@x.(a|b)+`
3. $P_3 =$ `f.@x+.(c|d)+.@x+.f`

The pattern `@x.a.b.c.@x` denotes the family of regular languages that consists of words in Σ^* with exactly 5 letters, the first one being equal to the last one, separated by `a.b.c`. A mobility pattern P denotes a regular language $\mathcal{L}(P) \subseteq \Gamma^*$. More generally a mobility pattern P with k variables is equivalent to the union of $|\Sigma|^k$ regular expressions enumerating the $|\Sigma|^k$ possible combinations of variables values. Variables give an exponentially concise way of expressing such languages.

In the following we shall denote as $var(P)$ the set of variables in a pattern P. The query language and its semantics are now defined as follows.

Definition 4 (Syntax of queries) *A* query *is a pair $(P, \mathcal{C})$ where P is a mobility pattern and $\mathcal{C}$ is a set of constraints of the form $s_1 \neq s_2$, for $s_1, s_2 \in \Sigma \cup var(P)$*

Let $q = (L, \{C_1, \dots C_l\})$ be a query. The answer to of q over $\mathcal{O}$, denoted $ans(q)$, is a subset of $\mathcal{O}$ defined as follows:

Definition 5 (Semantics of queries) *An object $o \in \mathcal{O}$ belongs to $ans(q)$ if there exists a mapping $\nu : \mathcal{V} \rightarrow \Sigma$, called a* valuation*, with the following properties:*

1. *ν satisfies all the constraints $C_i, i = 1, \dots, l$*
2. *$o.traj$ belongs to $\Sigma^*.\mathcal{L}(\nu(P))$.*

The constraints in a query can be used to forbid explicitely a variable to take a value (e.g., `@x` $\neq$ `a`). The *domain* of a variable `@x` for a given query q, denoted $dom_q($`@x`$)$, represents the set of possible values for `@x` given the constraints of q.

Example 2 *The following queries correspond to the 3 examples given in Section 2.*

1. $q_1 = (a.f\{2,\}.c, \emptyset)$
2. $q_2 = ((a|b)^+.@x.(a|b)^+, \{@x \neq a, @x \neq b\})$
3. $q_3 = (f.@x^+.(c|d)^+.@x^+.f, \{@x \neq f\})$

3.2 Query evaluation

We describe now an algorithm for evaluating a query q. First we show how to obtain an automaton which, given a mobility pattern P, accepts the trajectories that match P. This automaton also provides the valuation of variables in P. In a second step we explain how the automaton can be used at run time, and discuss the size of the memory used to store the relevant information. For simplicity, we consider the automata that accept the language $\mathcal{L}(P)$: their extension to automata that accept $\Sigma^*.\mathcal{L}(P)$ is trivial and can be found in any specialized textbook.

Since a mobility pattern P is a regular expression over the alphabet Γ, we can build a non-deterministic finite state automaton (NFA) N_Γ that accepts the language of Γ^* denoted by P. Starting from N_Γ we can build a new automaton, N_Σ, which checks whether a trajectory t of Σ^* belongs to $\nu(\mathcal{L}(P))$, and delivers the valuation ν.

Essentially, N_Σ is N_Γ with a management of variable bindings based on the following extensions: (i) a transition labeled with a variable @x on a symbol α sets the value of @x to α if @x was not yet bound and (ii) with each state one maintains the bindings of the variables met so far. Transitions from s_i to s_j, labeled with a variable @x, are then interpreted as follows:

1. If @x is bound to α in s_i, and the current symbol of the input word is α, then s_j can be reached and the binding of s_j is identical to the binding of s_i.
2. If @x is not bound in s_i, and the current symbol of the input word is α, then s_j can be reached and the binding of s_j is the binding of s_i augmented with $@x \leftarrow \alpha$.

The definition of N_Σ is as follows.

- The set of states of N_Σ, $states(N_\Sigma)$, is $states(N_\Gamma) \times \Sigma^{|var(P)|}$, i.e., all the possible associations of a state of N_Γ with a valuation ν of the variables in P. A state of N_Σ is denoted $< S, \nu >$.
- The set of accepting states of N_Σ, $accept(N_\Sigma)$ is $accept(N_\Gamma) \times \Sigma^{|var(P)|}$.
- The transition function of N_Σ, δ_Σ, is drawn from the transition function of N_Γ, δ_Γ, as follows:
  - if $\delta_\Gamma(S_i, \alpha) = S_j$ is a transition of N_Γ with $\alpha \in \Sigma$, then $\delta_\Sigma(< S_i, \nu > , \alpha) =< S_j, \nu >$. In other words the transition has no effect on variable bindings.

– if $\delta_\Gamma(S_i, @x) = S_j$ is a transition of N_Σ with $@x \in \mathcal{V}$, then $\delta_\Sigma(< S_i, \nu >, \alpha) =$

$$\begin{cases} < S_j, \nu + @x := \alpha > \\ \quad \text{if } \nu(@x) \text{ is undetermined and the binding} \\ \quad \text{of } @x \text{ with } \alpha \text{ is allowed by the constraints.} \\ < S_j, \nu > \quad \text{if } \nu(@x) = \alpha. \\ \text{is undefined otherwise.} \end{cases}$$

Whenever an accepting state $< S, \nu >$ of N_Σ is reached, the input trajectory is accepted and the valuation ν defines the instantiations of all the variables (recall that, by definition, any word in a language defined by a mobility pattern contains all the variables).

In order to check at run time whether an object o matches a mobility pattern, we do not need to fully construct the automaton described above. Instead, we start with a minimal representation, and build in a progressive way, according to the symbols appended to the trajectory of o, the instantiation of the variables which potentially leads to an accepting state. The initial representation of N_Σ consists only of the set of states of N_Γ, each associated with the empty valuation. By keeping all the current states of N_Σ associated with o, the following operations can be performed whenever a new move m is appended to $o.traj$ to test whether o enters, stays or quits the query result:

1. If the transitions labeled with m lead o to at least one accepting state, then o becomes part of the result of the query.

2. If the transitions labeled with m are such that o is no longer in at least one accepting state, then it must be removed from the result of the query.

This yields a first, convenient, property for the evaluation of continuous queries: the last move suffices to deliver the information needed to maintain a query result. Here is an example that illustrates the process (more details can be found in the long version).

Example 3 *Consider the mobility pattern $P = (a|b)^+.@x.(a|b)^+$. Figure 2 shows an NFA automaton N_Γ which recognizes the words of $\mathcal{L}(P)$, S_0 being the initial state and S_4, S_5 the final states.*

Assume that one receives successively the following events for an object o: a, a, b, b, c and a. Each row in the table of the figure 3 shows the states of the NFA N_Σ after reading a symbol, as well as the possible valuations of variable @x. The accepting states are in bold font and mean that the trajectory belongs to the query result set.

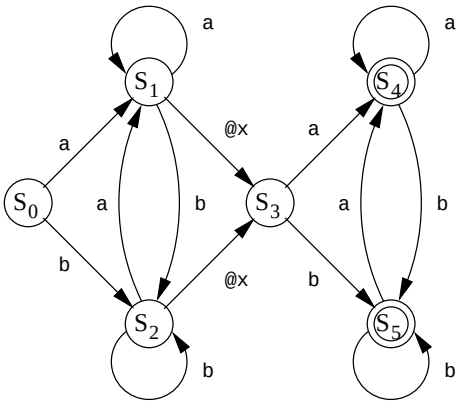

Figure 2: An automaton for the mobility pattern `(a|b)`$^+$`.@x.(a|b)`$^+$

Input	Reached states in N_Σ
a	$< S_1$,@x=$\perp >$
a[2]	$< S_1$,@x=$\perp >, < S_3$,@x=a$>$
a[2].**b**	$< S_2$,@x=$\perp >, < S_3$,@x=b$>$, $<$ **S$_5$**,**@x**=**a**$>$
a[2].**b**[2]	$< S_2$,@x=$\perp >, < S_3$,@x=$b >, <$ **S$_5$**,**@x**=**a**$>, <$ **S$_5$**,**@x**=**b**$>$
a[2].b[2].**c**	$< S_3$,@x=$c >$
a[2].b[2].c.**a**	$<$ **S$_4$**,**@x**=**c**$>$

Figure 3: Evaluation of a undeterministic query

Example 3 shows that we might have to maintain, during the analysis of an input trajectory, several valuations associated to a same state. In the worst case one might have $|states(N_\Gamma)| \times |\Sigma^k|$ simultaneous states to maintain, representing all the possible instantiations of variables that lead to an accepting state. Consider the following pattern:

$$@x + @y + @z+$$

It is no difficult to find a word such that $@x$, $@y$ and $@z$ take all their possible instantiations.

Depending on the application, the size of the database and the number of queries, maintaining a large amount of informations to continuously evaluate a query might become costly. In some cases we might therefore want to restrict the expressive power of the language to obtain very low memory needs. Consider for instance a web server providing a subscribe/publish mechanism over a (possibly large) set of moving objects. In such a system, web users can register queries, waiting for notification of the results. The performance of the system, and in particular its ability to serve a lot of queries under an intensive incoming of events,

depends on the efficiency of the query result maintenance, and therefore on the size of the data required to perform this maintenance. We define below a fragment of the query language which meets the requirement of this kind of application.

3.3 Deterministic queries

The class of *deterministic* queries is such that, at any instant, there is only one possible instantiation for each variable of the mobility patterns. Deterministic queries are defined by the following property:

Definition 6 (Deterministic queries) *A query $q(P, \mathcal{C}$ is* deterministic *iff* $\forall u, v \in (\Sigma \cup \mathcal{V})^*, \forall \texttt{@x} \in \mathcal{V}, u.\texttt{@x}.v \in \mathcal{L}(P) \Rightarrow \nexists \alpha \in dom_q(\texttt{@x}), \nexists w \in (\Sigma \cup \mathcal{V})^*, u\alpha w \in \mathcal{L}(P)$.

The intuition is that when it becomes possible to instantiate a variable during the analysis of a trajectory, then this transition is the only possible choice. This makes the binding of variables deterministic, and ensures that, for a given word, there is only one (if any) possibility to instantiate a variable.

Example 4 *The following examples illustrate deterministic queries.*

- *The query $q(\texttt{f.@x.(c|d).@x.f}, \emptyset)$ is deterministic. Whenever a* `f` *symbol has been read, the only possible choice is to bind* `@x` *to the symbol that follows immediately* `f`.
- *The query $q((a|b)^+.@x.(a|b)^+, \emptyset)$ is non-deterministic since the words* `a.@x.a.b` *and* `a.b.@x.b` *both belong to $\mathcal{L}(P)$. However $q'((a|b)^+.@x.(a|b)^+, \{@x \neq a, @x \neq b\})$ is deterministic.*

Proposition 2 *There exists an algorithm to check whether a query is deterministic.*

Proof (sketch): Let q be a query, P be a mobility pattern in q and N_Γ a deterministic automaton which recognizes $\mathcal{L}(P)$. Since N_Γ is deterministic, for any input string we reach at most one state s of N_Γ. If one can find a state s with two transitions: $\delta(s,\texttt{@x}) = s'$ and $\delta(s, \alpha) = s''$, with $\alpha \in dom_q(\texttt{@x})$, then it suffices to check whether there exists two words Xu and αv which both permit to reach a final state from s. If this is the case, then q is not deterministic. □

We state the following properties of deterministic queries without the proofs which can be found in the long version.

Proposition 3 *Let $q(P, \mathcal{C})$ be a deterministic query. Then, for each word w of Σ^*, there is at most one witness of w in $\mathcal{L}(P)$.*

Input	Reached states in N_Σ	Transitions not allowed
a	$< S_1$,@x=$\perp >$	
a[2]	$< S_1$,@x=$\perp >$	$< S_3$,@x=a> since a $\notin dom(@x)$
a[2].**b**	$< S_2$,@x=$\perp >$	$< S_3$,@x=b> since b $\notin dom(@x)$
a[2].**b**[2]	$< S_2$,@x=$\perp >$	$< S_3$,@x=b> since b $\notin dom(@x)$
a[2].b[2].**c**	$< S_3$,@x=c>	
a[2].b[2].c.**a**	$< \mathbf{S_4}$,**@x=c**>	

Figure 4: Evaluation of a deterministic query

Consider again the queries of Example 4. In the first example an accepted word can only have one single witness, either f.@x.d.@x.f or f.@x.c.@x.f. In the second example, with constraints {@x≠a,@x≠b}, any witness consists of two words of {a,b}$^+$, separated by a symbol distinct from a or b. It follows that if $q(P,\mathcal{C})$ is a deterministic query, the memory space required to check whether a word matches q is $|P| + |var(P)|$, where $|P|$ represents the number of symbols in P. Essentially, we need one FA for q, plus a storage for each variable, and we can build an FA with a number of states equal to the number of symbols in the expression.

When evaluating a continuous query, we need to maintain for each object o the set of its current states, as well as the binding of variables and this suffices to determine, at each GPS event, whether o enters, stays or quits the query result.

Example 5 *Let us consider again the query $q(P,\mathcal{C})$, with $P = (a|b)^+.@x.(a|b)^+$ and $\mathcal{C} = \{@x \neq a, @y \neq b\}$. The automaton remains identical (see Figure 2) but the evaluation on input* a[2].b[2].c.a *is now as presented in the table of the figure 4.*

The properties of deterministic queries ensure that the required amount of memory is independent from the size of Σ, and thus of the underlying partition of space used to describe the trajectories of moving objects. This property might be quite convenient if the space of interest is very large, or if the number of queries to maintain is such that the memory usage becomes a problem.

4 Conclusion and further work

We described in this paper a new approach for querying a moving object database by means of *mobility patterns*. Our proposal is based on a data model which allows to retrieve objects whose trajectory matches a parameterized sequence of moves

expressed with respect to a set of labeled zones. We investigated the applicability of the model to continuous query evaluation, showed how to maintain incrementally the result of a query, and identify a fragment of the query language such that the amount of space required to maintain this result is very low.

A version of the language can easily be introduced as complement of a geometric-based extension of SQL, as shown by the query samples proposed in Section 2. The properties of the language make it a convenient candidate for mobile object tracking based on sequences patterns, and its simplicity leads to an easy implementation.

We are currently developing a prototype to assess the relevancy of this approach in a web-based context where a lot of clients can register queries, receive an initial result set, and wait for notification of updates to this result set.

References

[1] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active Views for Electronic Commerce. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 1999.

[2] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–153, 1971.

[3] T. Brinkhoff and J. Weitkämper. Continuous Queries within an Architecture for Querying XML-Represented Moving Objects. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, 2001.

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[5] N. D., A. Fernandes, N. Paton, and T. Griffiths. Spatio-Temporal Evolution: Querying Patterns of Change in Spatio-Temporal Databases. In *Proc. Intl. Symp. on Geographic Information Systems*, pages 35–41, 2002.

[6] M. Dumas, M.-C. Fauvet, and P.-C. Scholl. Handling Temporal Grouping and Pattern-Matching Queries in a Temporal Object Model. In *Proc. Intl. Conf. on Information and Knowledge Management*, pages 424–431, 1998.

[7] F. Fabret, H. Jacobsen, F. Llirba, K. Ross, and D. Shasha. Filtering Algorithms and Implementations for Very Fast Publish/Subscrib Systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2001.

[8] L. Forlizzi, R. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[9] S. Grumbach, P. Rigaux, and L. Segoufi n. Manipulating Interpolated Data is Easier than you Thought. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2000.

[10] R. H. Gting, M. H. Bhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Quering Moving Objects. *ACM Trans. on Database Systems*, 25(1):1–42, 2000.

[11] M. Hammad, W. Aref, and A. Elmagarmid. Stream Window Join: Tracking Moving Objects in Sensor-Network Databases. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, 2003.

[12] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[13] D. Kalashnikov, S. Prabhakar, W. Aref, and S. Hambrusch. Effi cient evaluation of continuous range queries on moving objects. In *Proc. Intl. Conf. on Databases and Expert System Applications (DEXA)*, pages 731–740, 2002.

[14] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[15] G. Mecca and A. J. Bonner. Finite Query Languages for Sequence Databases. In *Proc. Intl. Workshop on Database Programming Languages*, 1995.

[16] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2000.

[17] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 84–95. IEEE Computer Society, 1998.

[18] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *Proc. ACM Symp. on Principles of Database Systems*, 2001.

[19] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. A sequential pattern query language for supporting instant data mining for e-services. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2001.

[20] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 232–239, 1995.

[21] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 422–433, 1997.

[22] A. P. Sistla, T. Hu, and V. Chowdhry. Similarity based retrieval from sequence databases using automata as queries. In *Proc. Intl. Conf. on Information and Knowledge Management*, pages 237–244, 2002.

[23] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 287–298, 2002.

[24] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1992.

[25] G. Trajcevski, P. Scheuermann, O. Wolfson, and N. Nedungadi. Cat: Correct answers of continuous queries using triggers. pages 837–840, 2004.

[26] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl (3rd Edition)*. O'Reilly, 2000.