INFORMATIK
BERICHTE

358 – 04/2011

Group Spatiotemporal Pattern Queries

Mahmoud A. Sakr, Ralf Hartmut Güting

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
Postfach 940
D-58084 Hagen

Group Spatiotemporal Pattern Queries

Mahmoud A. Sakr [#,*,1], Ralf Hartmut Güting [#,2]

[#] *Database Systems for New Applications, FernUniversität in Hagen*
58084 Hagen, Germany
[*] *Faculty of Computer and Information Sciences, University of Ain Shams*
Cairo, Egypt
[1] mahmoud.sakr@fernuni-hagen.de
[2] rhg@fernuni-hagen.de

April 5, 2011

Abstract

Reporting the motion patterns in moving objects data has been the focus of many research projects recently. Group spatiotemporal patterns are the movement patterns, in space and time, formed by groups of moving objects, such as flocks, concurrence, encounter, etc. There exist, in the literature, smart algorithms for matching some of these patterns. These solutions, however, address specific patterns and require specialized data representation and indexes. They share too little to be integrated into a single system. There is a need for a generic query method. In this paper, we propose a language that can consistently express and evaluate a wide range of group spatiotemporal pattern queries. We formally define the language operators, illustrate the evaluation algorithms, and discuss the optimization methods. Several examples are given to showcase the expressive power of the language.

1 Introduction

Spatiotemporal data are geometries that change their position and/or extent over time. Such kind of data exists everywhere around us (e.g. traces of vehicles, vegetation regions of certain plants, flocks of migrating birds, etc.). The advances in the positioning and sensor technologies have made it easier and cheaper to collect the traces of movement. The increasing amount of moving object data calls for analysis tools to better understand it. Therefore, spatiotemporal data mining is receiving a lot of attention recently.

Spatiotemporal pattern (STP) queries belong to this category of analysis tools. Generally they describe sets of changes/events that the moving objects conduct during their movement with some temporal order, and probably reflect interesting phenomena. According to the number of objects involved in the pattern, STPs might be individual or group. An example for individual STP queries is:
Find all trains that encountered a delay of more than half an hour after passing through a snow storm.
The example shows an STP that is a sequence of two predicates: *train crosses a snow storm*, and *delay of train* $>=$ *30 min*. The pattern can be completely evaluated for every *train*, hence the name *individual*. An individual STP query, hence, reports the moving objects that fulfill a set of predicates in a certain temporal order.

On the other hand, a *group STP query* reports the patterns that involve a collective movement of several moving objects, e.g.
Find a flock of geese migrating in northern direction.
The query looks for a group of geese traveling close to each other, in a flock, in northern direction. Answering such a query requires analysing the trajectories of several moving objects, and their spatiotemporal relationships to one another.

The individual STP queries were previously studied, and an expressive language was proposed in [17]. In this paper, we focus on group STP queries. The group STP queries are required for many application domains such as the behavioral study of animals, traffic monitoring, analyzing soccer games, etc. The topic has recently attracted many researchers as we discuss in Section 2. The techniques for reporting group STPs are either *online* (i.e. continuously searching for the pattern instances in a data stream), or *offline* (i.e. searching for pattern instances in the historical trajectories of the moving objects). This paper focuses on the *offline* pattern search.

This paper proposes an extensible language that is able to express and evaluate a wide range of group STP queries consistently. We also propose optimization methods for these queries. The proposed language design pays a lot of attention to the issues related to system integration. That is, the inputs and the outputs of the language operators are carefully designed to integrate with the query language of a spatiotemporal DBMS.

Most of the related work goes in the direction of providing algorithms for the efficient matching of specific STPs. With the exception of the work in [14], there are no proposals that offer such a language, up to our knowledge. This approach has, however, limitations that we discuss in detail.

The rest of this paper is organized as follows. Section 2 reviews the closely related work. Section 3 describes the moving objects representation that we assume in this paper. We give a non-formal illustration for the proposed language in Section 4. The formal definitions of the language operators follow in Sections 5, 6, and 7. The utility of the proposed language is demonstrated in Section 8 by several query examples. Section 9 describes the optimization of the language operators. Finally we conclude in Section 10, and tell about our future work plans.

2 Related Work

In their work, Dodge et al [5] presented a systematic taxonomy of movement patterns, along with an extensive survey of this area of research. They proposed a set of dimensions to classify the movement patterns, underlining the commonalities between them. Their work suggests that a generic toolbox for pattern search can be designed based on such a classification.

Andrienko and Andrienko have several publications that focus on solutions based on visual analytics, for example [3]. In particular, a set of data transformations, computations, and visualization techniques is defined, that could enable a human analyst to discover interesting patterns in large masses of moving objects data.

The algorithmic solutions of the problem tend to address very specific patterns. They share too little to be integrated within one system context. Moreover, there is no wide agreement on the definitions of the patterns. Table 1 lists some of these works. The table shows, for instance, two of the definitions of the *flock* pattern. The first is proposed in [15] and [8]. The pattern matching is done using the REMO approach, that we discuss below in detail. The second definition appears in [4]. The trajectories are represented as points in a high dimensional space, indexed by a *skip quad tree*. A flock query is then modeled as a count query to the skip quad tree. The *convoy* pattern [13] is another variant of the flock pattern.

The MoveMine application [16] implements several recent algorithms for group STPs, and trajectory clustering and classification. The application offers a GUI to set the parameters of every algorithm and to visualize the results. MoveMine did not show, however, efforts towards integrating these pattern matching algorithms in a unified computational framework.

Table 1: Examples for the Group STPs

1	*Concurrence* [14]: n moving point objects (MPO) showing the same motion attribute value (e.g. speed, azimuth) at time t.
2	*Trend-setter* [14]: One trend-setting MPO anticipates the motion of m others, so that they show the same motion attribute value.
3	(m, r) *Flock* [15], [8]: At least m MPOs are within a circular region of radius r and they move in the same direction.
4	(m, r, d) *Leadership* [15], [8]: At least m MPOs are within a circular region of radius r, and they move in the same direction. At least one of them was already moving in this direction for at least d time steps.
5	(m, r) *Convergence* [15], [8]: At least m MPOs will pass through the same circular region of radius r, assuming they keep their direction.
6	(m, r) *Encounter* [15], [8]: At least m MPOs will be simultaneously inside the same circular region of radius r (assuming they keep their speed and direction).
7	(m, r, d) *Convoy* [13]: At least d consecutive timesteps, during which an (m, r) *Density-based Cluster* is defined.
8	(m, r, d) *Flock* [4]: At least d consecutive timesteps, such that for every timestep there is a disk of radius r that contains all the m MPOs.

The work by Laube et al. [14] proposes the REMO (RElative MOtion) model. It is the most closely related work to this paper. It defines a language that can consistently express a number of group STP queries. Briefly, the model defines a 2D matrix, where the rows represent the moving objects, and the columns represent a series of time instants. Whereas the columns are clearly ordered by time, the rows have no inherent order. The elements of the matrix are the values of some predefined motion attribute (e.g. object's speed, azimuth, acceleration, etc.) computed for the moving object (row) at the time instant (column). The REMO matrix is hence a 2D string, where 2D string patterns can be searched for. The pattern query is expressed by means of regular expressions. The model could successfully express the *concurrence*, and the *trend-setter* patterns [14]. We see the following shortcomings in the REMO approach:
(1) The size of the REMO matrix is proportional to the database size. One would even need to maintain several matrices for several motion attributes.
(2) The REMO matrix does not inherently support the analysis based on the object locations (e.g. spatial proximity constraints between the moving objects). The model is extended in [15] to support such analysis functions as second class citizens. That is, first the analysis is done on the REMO matrix, then the analysis part that requires object locations is done on the results. For example, according to the REMO model a *flock* pattern is a *concurrence* pattern plus spatial proximity constraints. This would require that the flock members match the concurrence pattern (i.e. concurrently share similar values for the motion attributes). Such

a definition is clearly restrictive (e.g. cows in a flock may move around, without leaving their flock, with different speed and/or azimuth).
(3) The model cannot express the patterns that are described based on object interactions. The motion attributes in the REMO matrix describe every object independently of the other objects. Patterns that are described based on the mutual relationships between the moving objects (e.g. north of, closer than) cannot be expressed.
(4) The REMO matrix handles the time discretely. It does not directly support continuous trajectories. Trajectory sampling is known to incur inaccuracies in the data representation.

Up to our knowledge, this work by Laube et al. is the only one that proposes a generic language for group STP queries. The above analysis calls for a new approach that overcomes these problems.

3 The Underlying Data Model

This paper builds on the moving objects data model proposed in [12] and [7]. This section describes the parts of the model that will be used in the rest of the paper. We use the Second-Order Signature (SOS) [9] for the formal definitions. It is a tool for specifying data models, query processing, and optimization rules. It lets the user first define the type system (the first signature), then define polymorphic operations on the types (the second signature).

A signature consists of *sorts* and *operators*, and generates a set of *terms*. In the first signature, the one defining the type system, sorts are called *kinds*, operators are called *type constructors*, and terms are called *types*. That is, type constructors operate on kinds to generate types. We briefly describe the type system, and the query language of [12] and [7] in Sections 3.1 and 3.2. Then we extend this type system in Section 3.3.

3.1 The Base Type System

The upper part of Table 2 shows the type system of [7]. The bold parts are extensions that will be described in Section 3.3. The left, and middle columns display the argument and the result kinds, and the right column displays the type constructors. Kinds are sets of types. The kind DATA, for instance, contains four types. A type constructor that accepts no arguments is a constant, and yields a single type (e.g. $\underline{int}$, $\underline{point}$). A type constructor that accepts arguments generates a set, possibly infinite, of types (e.g. $\underline{range}(\underline{int})$, ..., $\underline{range}(\underline{instant})$).

The upper part of Table 2 defines abstract data types (ADT) for moving object representation, as was proposed in [7]. The lower part describes a relational data model, where ATTR = DATA $\cup$ ... $\cup$ MAPPING. We restrict ourselves in this paper to the relational model. It is however possible to introduce similar solutions for other database models (e.g. object relational, XML, etc.).

Table 2 defines the syntax of the type system. The semantics is defined by assigning a *domain* for every type. The base types (e.g. $\underline{int}$, $\underline{string}$, etc.) have similar domains as in programming languages, except that their domains here are extended by the value *undefined* (e.g. $D_{\underline{real}} = \mathbb{R} \cup \{undefined\}$, where D_{type} denotes the domain of $type$).

This type system defines two kinds for moving objects: UNIT and MAPPING. Together, the two kinds define the so-called *sliced representation* of moving objects [7]. That is, the complete movement of a moving object during a certain observation period is decomposed into *slices*, each of which describes the movement during a smaller time interval. An object of kind UNIT represents a single *slice*. An object of kind MAPPING is a set of UNITs/slices.

Let $D_{\underline{instant}}$ denote the domain of time instants $\underline{instant}$, isomorphic to $\mathbb{R}$. Let *Interval* be

Table 2: The Type System

	→ IDENT	*ident*
	→ DATA	*int*, *bool*, *real*, *string*
	→ DISCRETE	*int*, *bool*, *string*
	→ SPATIAL	*point*, *region*, *line*
	→ TIME	*instant*
DATA ∪ TIME	→ RANGE	*range*
DATA ∪ SPATIAL	→ **COLL**	***set***
DATA ∪ SPATIAL ∪ **COLL**	→ TEMPORAL	*intime*
DISCRETE ∪ **COLL**	→ UNIT	*constunit*
	→ UNIT	*ureal*, *upoint*, *uline*, *uregion*
UNIT	→ MAPPING	*mapping*
(*ident* × ATTR)$^+$	→ TUPLE	*tuple*
TUPLE	→ REL	*rel*
TUPLE	→ STREAM	*stream*

the set of time intervals defined as:

$$Interval = \{(t_1, t_2, lc, rc) |\ t_1, t_2 \in D_{\underline{instant}},\ lc, rc \in \{\text{false, true}\},\ t_1 \leq t_2,\ (t_1 = t_2) \Rightarrow lc = rc = \text{true}\}$$

That is, a time interval can be left-closed and/or right-closed as indicated by the values of lc and rc respectively. It is also possible that the interval collapses into a single time instant.

A type in the UNIT kind describes a pair consisting of a time interval and a temporal function. The temporal function describes the evolution of the value of the moving object during the associated time interval. For the types that have discrete domains (i.e. DISCRETE ∪ COLL), the temporal function is a constant, and the type constructor *constunit* constructs their unit types. Let D_σ be the domain of a type $\sigma \in$ DISCRETE ∪ COLL. The domain of the corresponding unit type is:

$$D_{\underline{constunit}(\sigma)} = Interval \times D_\sigma$$

For example, the domain $D_{\underline{constunit}(\underline{bool})} = Interval \times \{\text{false, true,undefined}\}$. For the types that have continuous domains (e.g. *real*, *point*), the domains of their unit types are defined individually. For example, the domain of the unit point is:

$$D_{\underline{upoint}} = Interval \times (\mathbb{R}^2 \times \mathbb{R}^2)$$

The unit point describes a linearly moving point in the form $(I, ((x_1, y_1), (x_2, y_2)))$. The position of the point at time $t \in I$ is

$$(x_1 + \frac{(x_2 - x_1)(t - I.t_1)}{I.t_2 - I.t_1},\ y_1 + \frac{(y_2 - y_1)(t - I.t_1)}{I.t_2 - I.t_1})$$

A type in the MAPPING kind describes the complete movement of a moving object during some observation periods. It is therefore represented as a set of UNITs.

Definition 1 *$\forall\, unit$ in UNIT, let D_{unit} denote its domain. The domain of the type $\underline{mapping}(unit)$ is:*

$$
\begin{aligned}
D_{\underline{mapping}(unit)} = \{U \subset D_{unit} \mid \forall (i_1, f_1) \in U, (i_2, f_2) \in U : \\
(i)\ i_1 = i_2 \Rightarrow f_1 = f_2 \\
(ii)\ i_1 \neq i_2 \Rightarrow (i_1 \cap i_2 = \emptyset) \wedge \\
(i_1 \text{ adjacent } i_2 \ \Rightarrow\ \neg((i_1, f_1) \text{ mergeable } (i_2, f_2)))\}
\end{aligned}
$$

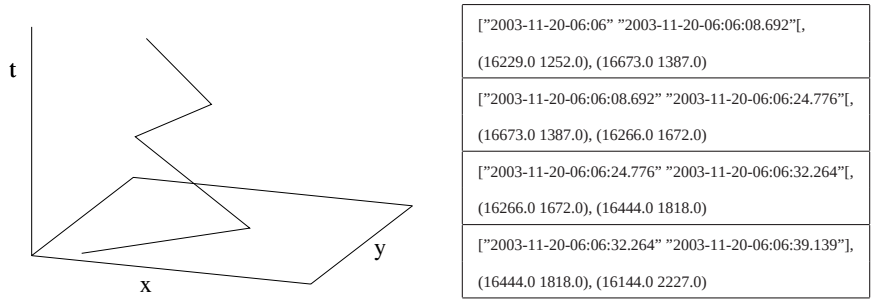
□

where i_1 *adjacent* $i_2 :\Leftrightarrow (i_1.t_2 = i_2.t_1) \vee (i_1.t_1 = i_2.t_2)$, and *mergeable* yields true if the two units can be merged together. It is defined according to the unit type. For constant units ($\underline{constunit}$):

$$((i_1, f_1)\ mergeable\ (i_2, f_2)) = (f_1 = f_2).$$

For $\underline{upoint}$, for instance, *mergeable* yields true if both units have the same direction and speed, and share one end point. This last condition ensures that a moving object has a unique representation, the one with the minimum number of units. Note that the definition allows for temporal gaps during which the moving object is undefined. In the following, the types in MAPPING are denoted by a preceding *m* (e.g. $\underline{mint}$, $\underline{mpoint}$, $\underline{mbool}$). Figure 1 illustrates an $\underline{mpoint}$ object.

Figure 1: The sliced representation of an $\underline{mpoint}$



3.2 The Query Language

A large number of operations is defined for the type system described above, see [7] and [12]. They fall into three classes:

1. Static operations defined on the non-temporal types (e.g. topological predicates, set operations, aggregations).

2. Spatiotemporal operations offered for the temporal types (e.g. trajectory of an $\underline{mpoint}$, evaluation of an $\underline{mregion}$ for a given instant of time).

3. Lifted operations offered for combinations of moving and non-moving types. Basically they are time-dependent versions of the static operations.

A large part of this moving objects model is implemented in the SECONDO system [1], [10]. It consists of three modules: the kernel, the query optimizer, and the graphical user interface.

The SECONDO kernel accepts queries in a language called the SECONDO *executable language*. It is a procedural language, in which one specifies step-by-step methods for achieving

the results. It consists of the three classes of operations above and of query processing operations of the relational model, usually applied in a stream processing mode. Essentially it is a precisely defined notation for query plans.

Whereas other database systems represent query plans as operator trees generated by the optimizer, SECONDO to our knowledge is unique in offering a complete syntax for query plans and a corresponding language level that is accessible to the user. In other words, the user can type query plans directly; these are parsed, checked for correct composition of operations, and then executed.

The SECONDO optimizer offers an SQL-like language like other database systems. The three classes of operations mentioned above are integrated into this level as well. The optimizer uses cost based optimization to map SQL queries into query plans of the SECONDO executable language. Any plan that the optimizer generates could as well be typed directly by a user.

The language for group spatiotemporal pattern queries we describe in this paper requires extensions to a database system both at the levels of query processing and of query optimization. The first step is to extend query processing by new operators. The second step is to also extend the optimizer to make use of the new query processing operators and to apply further optimizations.

In the sequel, we will first embed our proposed language into the SECONDO executable language. Since this language level is accessible to the user, we will explain the concepts at this level and formulate also example queries at this level, making use of SECONDO query processing operations as needed. This allows us to explain the language in terms of precisely defined query processing operations and their related algorithms, without being bothered by the additional level of complexity resulting from embedding into SQL and query optimization. Later in Section 9 we explain the second step of integrating the language operations into the SQL language on top of the optimizer.

Here we briefly introduce the executable language level. Let *PhoneBook* be a relation with the type: $\underline{rel}(\underline{tuple}(<$(Name, $\underline{string}$), (Phone, $\underline{string}$)$>))$. A query that finds the entries with the name *Ali Mahmoud* is:

```
query PhoneBook feed
  filter[.Name contains "Ali Mahmoud"] consume;
```

The *feed* operator reads a relation from disk and converts it into a tuple stream. The *consume* operator does the opposite. The *filter* operator evaluates a boolean expression for every input tuple, and yields the tuples that fulfill it. For such stream processing operations usually a postfix syntax is defined so that one can conveniently write query operators in the order in which they are applied. The signatures of these operators are:

$\underline{rel}(tuple)$	$\rightarrow \underline{stream}(tuple)$	**feed**	_ #
$\underline{stream}(tuple)$	$\rightarrow \underline{rel}(tuple)$	**consume**	_ #
$\underline{stream}(tuple) \times (tuple \rightarrow \underline{bool})$	$\rightarrow \underline{stream}(tuple)$	**filter**	_ #[_]
$\underline{string} \times \underline{string}$	$\rightarrow \underline{bool}$	**contains**	_ # _

where *tuple* is a type variable that can be instantiated with any type in TUPLE. The last column in the signature shows the operator syntax, where # denotes the operator, and _ denotes an argument. We will describe more operations in the sequel.

3.3 Extending the Type System

We extend the type system in [7] by the kind COLL as shown in bold in Table 2. This is to introduce the moving set type $\underline{mset}$ (i.e. $\underline{mset}$ denotes $\underline{mapping}(\underline{constunit}(\underline{set}))$). It represents a set whose elements are changing over time. The domain $D_{\underline{mset}}$ of the type $\underline{mset}$ can

be obtained by applying Definition 1. That is, let *elem* be a type variable that can be instantiated by any type in DATA ∪ SPATIAL, and let D_{elem} be its domain. The domain of the type $\underline{set}$ is $D_{\underline{set}} = \mathcal{P}(D_{elem})$, and the domain of the type $\underline{constunit}(\underline{set})$ is $D_{\underline{constunit}(\underline{set})} = Interval \times D_{\underline{set}}$. Finally $D_{\underline{mset}}$ is obtained from Definition 1 as $D_{\underline{mapping}(\underline{constunit}(\underline{set}))}$. The type $\underline{mset}$ is used to represent the groups of moving objects that match a group STP query, as will be shown in Section 4. We also define the set *MSetPart*, which is required for the operator definitions, as follows:

Definition 2 *Let $MSetPart$ be a subset of $D_{\underline{mset}}$ defined as:*

$$MSetPart = \{\ \{(i_1, f_1), ..., (i_n, f_n)\} \in D_{\underline{mset}} | \\ \exists\, i \in Interval : i = (i_1 \cup ... \cup i_n)\}$$

□

That is, an element of the $MSetPart$ is an $\underline{mset}$ instance that has no definition gaps within its definition time. Its definition time can be represented as a single time interval.

4 The Proposed Language

This section roughly illustrates the proposed language. The details of the operators, their formal definitions, and the evaluation algorithms follow in Sections 5, 6, and 7, respectively. The proposed language is a nested operator structure having three levels. In the bottom-most level are the time-dependent predicates. These are operations that yield $\underline{mbool}$ values. They are part of the model in [12] and [7] that we described in Section 3. Figure 2 illustrates the time-dependent predicate *inside*, whose signature is:

$$\underline{mpoint} \times \underline{region} \qquad \rightarrow \underline{mbool} \qquad \textbf{inside} \qquad \_ \# \_$$

It yields true in the time intervals during which the $\underline{mpoint}$ object is spatially located within the $\underline{region}$ object, false otherwise.

The group STPs are typically expressed based on some motion attributes (e.g. direction, area, distance between objects, etc.). These attributes can be the raw coordinates (e.g. find cars meeting in down town), a derivative (e.g. find a cattle moving with high speed), or a second derivative (e.g. find oil spills increasing their area). Time-dependent predicates are able to express conditions on such attributes. They are also defined for all kinds of moving objects (e.g. $\underline{mpoint}$, $\underline{mregion}$, $\underline{mreal}$). We express the group STP queries on top of them, hence allowing for complex analysis on all kinds of motion attributes and all types of moving objects.

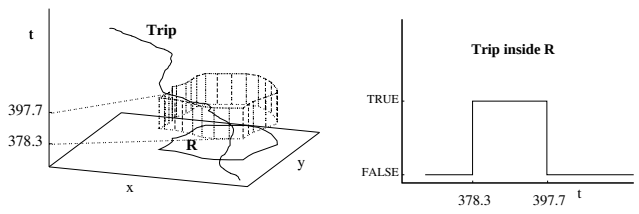


Figure 2: Time-dependent Predicates

In the second level are the *patternoid* operators. A patternoid is a simple group STP such as *flock*, *convergence*, *concurrence*, etc. This is in contrast to the composite group STPs which

consist of several patternoids (e.g. a *convergence* followed by a *flock*). In this paper, we describe two patternoid operators, the *gpattern* and the *crosspattern*. Both of them are built on top of the time-dependent predicates. The *gpattern* operator expresses the patternoids in terms of independently computed motion attributes of the moving objects. The *concurrence* patternoid, for instance, is a group of objects that concurrently fulfill some time-dependent predicate. The time-dependent predicate is evaluated for every moving object independently from other objects.

The *crosspattern* operator expresses the patternoids that can be described in terms of dual interactions between the moving objects (e.g. *flock, convoy, convergence, etc.*). A flock, for instance, is expressed in terms of the distance between pairs of moving objects. The *crosspattern* operator, in contrast to the *gpattern* operator, evaluates the time-dependent predicate for pairs of moving objects.

The *reportpattern* is the top-most operator at the third level. It allows for expressing composite group STPs. It allows one to specify constraints (e.g. temporal and spatial) between a list of patternoids. In the following, we show three examples that illustrate the *gpattern*, the *crosspattern*, and the *reportpattern* operators, respectively. We assume the schema:

Gazelles[Id: $\underline{int}$, Trip: $\underline{mpoint}$]

Example 1 Find a group of 20 gazelles that move concurrently with a speed of more than 20 km/h, for at least 10 minutes.

```
let ten_minutes= create_duration(0, 600000);
query Gazelles feed
  gpattern[.Id, speed(.Trip) > 20.0, ten_minutes, 20, "atleast"]
  transformstream consume;
```

The *gpattern* operator gets a stream of tuples. It evaluates the time-dependent predicate *speed(.Trip) > 20.0* for every tuple in the stream. Using the evaluated $\underline{mbool}$s, it finds the groups with cardinality of *at least 20* gazelles, that concurrently fulfill the time-dependent predicate, for at least 10 minutes. The result of the *gpattern* operator is represented as a $\underline{stream}(\underline{mset})$, as will be described later in detail. Within the results, gazelles are represented by their identifiers, which are supplied by the argument *.Id*. The *transformstream* casts the $\underline{stream}(\underline{mset})$ result of *gpattern* into a $\underline{stream}(\underline{tuple}(<(\text{elem}, \underline{mset})>))$, so that it can be passed to the *consume* operator.

The following example illustrates the *crosspattern* operator.

Example 2 Find a group of at least 20 gazelles moving simultaneously for 10 minutes towards some place of meeting.

The example describes a relaxed version of the *convergence* patternoid in Table 1. The definition in the table requires that the gazelles keep their movement direction (azimuth) to the meeting place unchanged. Gazelles that change their direction, to maneuver around some obstacle, for instance, will not be reported. In the following query, we express convergence in terms of the continuous decrease in the dual distance between the gazelles.

```
query Gazelles feed {a} Gazelles feed {b}
  symmjoin[.Id_a < ..Id_b]
  crosspattern[.Id_a, .Id_b,
    isdecreasing(distance(.Trip_a, .Trip_b)),
    ten_minutes, 20, "clique" )]
  transformstream consume;
```

Since the *crosspattern* operator requires pairs of gazelles in the input stream, the *Gazelles* relation is first self-joined by the *symmjoin* operator. The *.Id_a* and *.Id_b* arguments of the

crosspattern operator tell it about the identifiers of the pair of gazelles. The time-dependent predicate is evaluated for every input tuple. This computes a kind of *time-dependent join*. That is, two tuples from the original *Gazelles* relation join together whenever the time-dependent predicate holds.

This join result is represented as a time-dependent graph, as will be defined in Section 7. The nodes of this graph are the gazelles (i.e. every node corresponds to an identifier of a gazelle), and the time-dependent edges are the evaluated $\underline{mbool}$s. An edge exists between two nodes, whenever the $\underline{mbool}$ is true (i.e. the two gazelles are getting closer to each other).

The last three arguments in the *crosspattern* operator define search criteria for a special subgraph within this time-dependent graph. In this example, we look for a *clique* containing at least 20 nodes, that lasts for a duration of at least ten minutes. A *clique* in this case means that every gazelle in the clique is continuously getting closer to every other gazelle in the clique. One can specify other sub-graph types according to the patternoid one is describing (e.g. *a connected component*).

The self-join of the Gazelles relation in this example is a brute force solution to obtain pairs of gazelles. For an efficient execution, one would use an index to retrieve only the pairs of gazelles that have chances to fulfill the time-dependent predicate. In Section 9.2, we propose an approach to integrate the *crosspattern* operator with the query optimizer.

The following example illustrates the *reportpattern* operator.

Example 3 Find a group of at least 20 gazelles which moved simultaneously for 10 minutes towards some place of meeting, then moved together with a speed of at least 20 km/h.

The query describes a composite pattern that consists of the two patternoids in the previous examples. They are composed together based on three constraints: (1) a temporal constraint restricting the concurrence patternoid to occur after the convergence patternoid, (2) a spatial constraint that the concurrence patternoid starts from the final location of the convergence patternoid (i.e. the place of meeting), (3) and an attribute constraint that the same group of moving objects matches the two patternoids.The query looks as follows:

```
let then = vec("abab", "aa.bb", "aabb");
query reportpattern[
    concurrence: Gazelles feed
      gpattern[.Id, speed(.Trip) > 20.0, ten_minutes, 20, "atleast"],
    converge: Gazelles feed {g1} Gazelles feed {g2}
      symmjoin[.Id_g1 < ..Id_g2]
      crosspattern[.Id_g1, .Id_g2,
        isdecreasing(distance(Trip_g1, Trip_g2)),
        ten_minutes, 20, "clique" ];
    stconstraint(.converge, .concurrence, then)]
  extend[
    concMReg: fun(t1: TUPLE) Gazelles feed
      mset2mreg(.Id, .Trip, attr(t1, concurrence)),
    convMReg: fun(t2: TUPLE) Gazelles feed
      mset2mreg(.Id, .Trip, attr(t2,.converge)]
  extend[
    convFinSet: val(final(.converge))
    concInitSet: val(initial(.concurrence))
    convFinReg: val(final(.convMReg))
    concInitReg: val(initial(.concMReg))]
  filter[.convFinSet intersection .concInitSet count >= 20]
  filter[.convFinReg intersects .concInitReg]
  filter[not(sometimes(area(.concMReg)> 8000.0)]
  consume;
```

The *reportpattern* operator in this query contains two patternoid operators, and one temporal constraint. The two patternoids are given the aliases *concurrence* and *converge*. These aliases are required to refer to the patternoids in the rest of the query. This is analogous to attribute aliases in standard SQL. The temporal constraint is expressed by the *stconstraint* operator. It states that *converge* happens first *then concurrence*, where *then* is defined by the *let* command before the query.

Each of the terms *abab*, *aa.bb*, *aabb* that define *then* describes a relationship between two time intervals. The start and the end time instants of the first interval are denoted *aa*, and those of the second interval are denoted *bb*. The order of the symbols describes the interval relationship visually. The dot symbol denotes the equality. For example, the relation *aa.bb* between the intervals i_1, i_2 denotes the order: $((i_1.t_1 < i_1.t_2) \wedge (i_1.t_2 = i_2.t_1) \wedge (i_2.t_1 < i_2.t_2))$. The temporal relationship *then* expresses the disjunction of its three components.

Each of the two patternoid operators yields a $\underline{stream}(\underline{mset})$. Every $\underline{mset}$ represents a group of moving objects that matches the patternoid. Note that the group members may be changing over time, because moving objects may be joining or leaving the group, as long as the group cardinality does not go below the threshold. Therefore, a patternoid is represented as an $\underline{mset}$ rather than a $\underline{set}$.

The *reportpattern* operator computes the product of the $\underline{mset}$ streams coming from the patternoid operators, and filters it using the temporal constraints. In this example, a pair of $\underline{mset}$s fulfill the temporal constraint if their definition times fulfill the *then* relationship (i.e. by fulfilling any of its three components). As will be explained in Section 5, an $\underline{mset}$ in the result of a patternoid operator belongs to the set *MSetPart*. This is to guarantee that its definition time is a single time interval, hence temporal constraints can be applied to pairs of them. The *reportpattern* operator in this query yields the a tuple stream with the type (schema):

$$\underline{stream}(\underline{tuple}(< (\text{concurrence}, \underline{mset}), (\text{converge}, \underline{mset}) >))$$

The spatial and the attribute constraints between the two patternoids are applied to this tuple stream. The first *extend* operator adds two attributes: *concMReg* and *convMReg*. They are $\underline{mregion}$ representations of the *concurrence* and the *converge* patternoids. Each of them represents the convex-hull of the gazelle locations at every time instant. Now the tuple stream has the schema:

$$\begin{aligned}\underline{stream}(\underline{tuple}(< &(\text{concurrence}, \underline{mset}), (\text{converge}, \underline{mset}),\\ &(\text{concMReg}, \underline{mregion}), (\text{convMReg}, \underline{mregion}) >))\end{aligned}$$

The second *extend* operator adds four attributes, namely *convFinSet* and *concInitSet*, the final and initial values of the *converge* and *concurrence* $\underline{mset}$s, and *convFinReg* and *concInitReg*, the final and initial values of the corresponding moving region representations created in the previous step. At this point, the schema is:

$$\begin{aligned}\underline{stream}(\underline{tuple}(< &(\text{concurrence}, \underline{mset}), (\text{converge}, \underline{mset}),\\ &(\text{concMReg}, \underline{mregion}), (\text{convMReg}, \underline{mregion}),\\ &(\text{convFinSet}, \underline{set}), (\text{concInitSet}, \underline{set}),\\ &(\text{convFinReg}, \underline{region}), (\text{convMReg}, \underline{region}) >))\end{aligned}$$

The attribute constraint is expressed by the first *filter* operator in the query. It enforces that the set of gazelles at the final time instant of *converge* and the set of gazelles at the initial time instant of *concurrence* have at least 20 gazelles in common. The spatial constraint is expressed

Table 3: The Proposed Language

$(\underline{string})^+ \rightarrow \underline{stvector}$	**vec**	$\#(\_)$
$tuple \times (tuple \rightarrow \underline{interval})^2 \times \underline{stvector} \rightarrow \underline{bool}$	**stconstraint**	$\_\ \#(\_,\ \_,\ \_)$
$\underline{stream}(tuple) \times (tuple \rightarrow \underline{int}) \times (tuple \rightarrow \underline{mpoint})$ $\times \underline{mset} \rightarrow \underline{mregion}$	**mset2mreg**	$\#(\_,\ \_,\ \_,\ \_)$
$\underline{stream}(tuple) \times (tuple \rightarrow \underline{int}) \times (tuple \rightarrow \underline{mpoint})$ $\times \underline{mset} \rightarrow \underline{stream}(\underline{mpoint})$	**mset2mpoints**	$\#(\_,\ \_,\ \_,\ \_)$
$\underline{stream}(tuple) \times (tuple \rightarrow \underline{int}) \times (tuple \rightarrow \underline{mbool})$ $\times \underline{duration} \times \underline{int} \times \underline{string} \rightarrow \underline{stream}(\underline{mset})$	**gpattern**	$\_\ \#[\_,\ \_,\ \_,\ \_,\ \_]$
$\underline{stream}(tuple) \times (tuple \rightarrow \underline{int})^2 \times (tuple \rightarrow \underline{mbool})$ $\times \underline{duration} \times \underline{int} \times \underline{string} \rightarrow \underline{stream}(\underline{mset})$	**crosspattern**	$\_\ \#[\_,\ \_,\ \_,\ \_,\ \_,\ \_]$
$(\underline{ident} \times \underline{stream}(\underline{mset}))^+ \times (tuple \rightarrow \underline{bool})^+$ $\rightarrow \underline{stream}(\underline{tuple}(< (\underline{ident}, \underline{mset})^+ >))$	**reportpattern**	$\#[\_;\ \_]$

by the second *filter* operator. It enforces that the region representing *converge* at its final time instant, and the region representing *concurrence* at its initial time instant intersect.

The last *filter* operator constrains the gazelles in *concurrence* to keep close to one another in a circle of area 8000 m^2. We added this to illustrate how the spatial proximity constraints can be applied to the results of the *gpattern* operator in a way similar to [15].

Table 3 lists the signatures of the operators that constitute the proposed language. The $\underline{stvector}$ is a type that represents the temporal relationships, such as *then*, in a compact integer representation. The $\underline{interval}$ type represents a time interval. The following sections describe these operators formally.

5 The Reportpattern Operator

In this section, we formally define the *reportpattern* operator. It gets two lists: a list of patternoid operators each of which has an alias, and a list of temporal constraints. Syntactically, the *reportpattern* operator requires that a patternoid operator yields a $\underline{stream}(\underline{mset})$, and that a temporal constraint is a mapping $(tuple \rightarrow \underline{bool})$. Semantically, the result of a patternoid operator is required be a subset of $MSetPart$. This is required to be able to evaluate the temporal constraints, as will be explained below in this Section.

We start by defining a language for the temporal constraints. A similar language was defined in [17], in the context of individual STP queries. Given two time intervals, each of which may degenerate into a time instant, there are 26 possible relationships between them. We define a set IR consisting of 26 terms, each of which expresses one such relationship. That is:

$$IR = \{\text{aabb, abba, bbaa, a.bab, aa.bb, a.bba, bb.aa, baa.b,}$$
$$\text{abab, aba.b, baba, a.ba.b, baab, a.abb, bb.a.a, a.a.bb, bba.a,}$$
$$\text{ba.ab, b.baa, aa.b.b, b.b.aa, aab.b, ab.ba, a.ab.b, b.ba.a, a.a.b.b}\}$$

where a term in IR is formally defined as follows. Let $i_1, i_2 \in Interval$, $ir = s_1 s_2 ... s_k \in IR$,

$$\text{Let } rep(s_j) = \begin{cases} i_1.t_1 & \text{if } s_j \text{ is the first } \textit{a} \text{ in } ir \\ i_1.t_2 & \text{if } s_j \text{ is the second } \textit{a} \text{ in } ir \\ i_2.t_1 & \text{if } s_j \text{ is the first } \textit{b} \text{ in } ir \\ i_2.t_2 & \text{if } s_j \text{ is the second } \textit{b} \text{ in } ir \\ . & \text{if } s_j = . \end{cases}$$

$$i_1 \text{ and } i_2 \text{ fulfill } s_1 s_2 ... s_k \;\; :\Leftrightarrow \forall j \in \{1, ..., k-1\} :$$
$$(i) s_j \neq . \neq s_{j+1} \Rightarrow rep(s_j) < rep(s_{j+1})$$
$$(ii) s_{j+1} = . \Rightarrow rep(s_j) = rep(s_{j+2})$$

Two time intervals $i_1, i_2 \in Interval$ fulfill a *set* of interval relationships if they fulfill any of them, that is:

$$i_1 \text{ and } i_2 \text{ fulfill } SI \subseteq IR :\Leftrightarrow \exists\, ir \in SI :\; i_1 \text{ and } i_2 \text{ fulfill } ir$$

Syntactically, the *vec* operator in Table 3 allows for composing such SI subsets, and representing them as an object of type $\underline{stvector}$.

Let $P = \{p_1, ..., p_n\}$ be a set of patternoid operators. A temporal constraint on P is an element of the set:

$$TC(P) = \{1..n\} \times \{1..n\} \times \mathcal{P}(IR)$$

It is hence a binary constraint that assigns a pair of patternoid operators from P a set of interval relationships. Syntactically, it is expressed by the *stconstraint* operator in Table 3.

Let $P = \{p_1, ..., p_n\}$ be a set of patternoid operators. Let *eval*(p_i) denote the evaluation of the patternoid operator $p_i \in P$, that is, *eval*(p_i) $\subset$ *MSetPart*. We define the set of *candidate assignments* $CA(P)$ as:

$$CA(P) = eval(p_1) \times ... \times eval(p_n)$$

That is, the $CA(P)$ is simply the Cartesian product of the result streams of the patternoid operators.

Let $ca = (v_1, ..., v_n) \in CA(P)$ and let $c = (j, k, SI) \in TC(P)$ be a temporal constraint.

$$ca \;\text{ fulfills }\; c :\Leftrightarrow deftime(v_j) \;\text{ and }\; deftime(v_k) \;\text{ fulfill }\; SI$$

where $deftime(v_i)$ is the time interval during which the $\underline{mset}$ v_i is defined. Since the *reportpattern* operator requires that a result from a patternoid operator belongs to *MSetPart*, it is guaranteed that $deftime(v_i)$ yields a single time interval.

Let $C \subseteq TC(P)$ be a set of temporal constraints. The set of *supported assignments* of C is defined as:

$$SA(P, C) = \{ca \in \; CA(P) \;\mid \forall\, c \in \; C : ca \;\text{ fulfills }\; c\}$$

That is, for a *candidate assignment* to be a *supported assignment*, it must fulfill all the constraints in C. A supported assignment, hence, contains a single $\underline{mset}$ instance for every patternoid, and fulfills all the temporal constraints. The result of the *reportpattern* operator is the set of supported assignments. That is:

Definition 3 *The* reportpattern *operator is a pair (P, C), where $P = \{p_1, ..., p_n\}$ is a set of patternoid operators, and $C \subseteq TC(P)$ is a set of temporal constraints. Its evaluation is defined as:*

$$eval((P, C)) = SA(P, C)$$

□

In order to evaluate the *reportpattern* operator, we use the classical model of the *constraint satisfaction problem CSP*. A similar approach was adopted in [17]. A CSP is a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a set of the initial domains of X, and C is a set of constraints.

The solution of the CSP is a set of tuples, each of which contains one value from the domain of each variable, and fulfills all the constraints in C. Such a tuple is called a *supported assignment* of the CSP.

The definitions of the *reportpattern* operator and the CSP map to one another. The CSP variables X are the patternoid operators. The domain D_i of a variable X_i consists of the $\underline{mset}$s that result when evaluating the corresponding patternoid operator. The CSP constraints are the temporal constraints in the *reportpattern* operator.

This CSP is solved incrementally, as illustrated in Algorithm 1. The algorithm solves the CSP_{k-1} (i.e. having $k-1$ variables) first, then extends it to the CSP_k. Note that evaluating the domain of a variable is equivalent to evaluating a patternoid operator. Since such an evaluation is expected to be expensive, we wish to minimize the number of evaluated CSP variable domains, in the case that no solutions exist. The incremental evaluation allows for an early stop if a solution to the CSP_{k-1} cannot be found, avoiding the unnecessary evaluation of the remaining patternoid operators.

The *Agenda* stores the patternoid operators (i.e. the variables), that are not yet evaluated. In every iteration, one patternoid operator is selected from the Agenda and evaluated by the *Pick* function. The selection heuristic tries to discover as soon as possible whether the CSP has no solutions. It selects the variable from the Agenda that leads to the evaluation of the largest number of constraints in C during the current iteration. The same selection method was adopted in [17].

The sub-CSP, that consists of all the patternoid operators that are evaluated so far, is solved by the *extend* function in Algorithm 2. Only consistent solutions (i.e. patternoids that fulfill the temporal constraints) remain in the supported assignments set *SA*. The algorithm stops immediately and returns an empty stream once the set *SA* is empty. A temporal constraint, as illustrated in Table 3, accepts a *tuple* containing as many attributes as the number of patternoid operators. This tuple is dynamically generated by the *reportpattern* operator during execution, and passed as a parameter to the *stconstraint* operator, as illustrated in lines 10 and 12 in Algorithm 2. The attribute names within it are the aliases of the patternoid operators. The attributes values are the definition times of the patternoids in a candidate assignment. The *stconstraint* operator assigns two attributes from this input tuple a set of interval relationships composed by the *vec* operator.

Algorithm 1: *reportpattern*

Input: *P*: ($\underline{ident}$, $\underline{stream}(\underline{mset})$)$^+$ – a set of patternoid operators, *C*: (*tuple* $\rightarrow$ $\underline{bool}$)$^+$ – a set of temporal constraints

Output: *R*: $\underline{stream}(\underline{tuple}(<(\underline{ident}, \underline{mset})^+>))$

```
1  let SA: list(list(mset)) be a list of supported assignments, initially empty;
2  let Agenda := P;
3  while Agenda is not empty do
4  |  i := pick p_i from Agenda ; // select and remove one variable from
   |  the Agenda
5  |  let d := evaluate p_i.second;
6  |  extend(SA, d, i, C);
7  |  if SA is empty then
8  |  |  return an empty stream;
9  let R := construct the result stream from SA;
10 return R;
```

Algorithm 2: extend(*SA*: $\underline{list}(\underline{list}(\underline{mset}))$, *d*: $\underline{stream}(\underline{mset})$), *i*: $\underline{int}$, *C*: (*tuple* $\rightarrow \underline{bool})^+$)

```
1  if SA is empty then
2  |  foreach v: mset in d do
3  |  |  insert a new row sa into SA having sa[i] = v;
4  else
5  |  let CA be a list of candidate assignments, initially empty;
6  |  foreach sa in SA do
7  |  |  foreach v: mset in d do
8  |  |  |  let ca := sa;
9  |  |  |  ca[i] := v;
10 |  |  |  let ca_tuple := construct a tuple from ca;
11 |  |  |  foreach c: (tuple → bool) in C do
12 |  |  |  |  SetArgument(c, ca_tuple);
13 |  |  |  |  if ca_tuple fulfills c then insert ca into CA;
14 |  SA := CA;
```

6 The Gpattern Operator

This section formally defines the *gpattern* operator. Mainly it reports groups of moving objects that simultaneously fulfill a time-dependent predicate. It has two forms:

- S gpattern[id, α, d, n, "*exactly*"].
- S gpattern[id, α, d, n, "*atleast*"].

where S is a stream of tuples representing the whole set of moving objects. It contains a moving object attribute, and an identifier attribute, which is required for a technical reason, to represent the moving object within the results. id is a function that maps a tuple in S into the value of this identifier attribute, α is the time-dependent predicate, $d > 0$ is a $\underline{duration}$ (e.g. in milliseconds) that decides the minimum duration of the patternoid, and $n > 0$ is an $\underline{int}$ that decides the size of the group. In the following definitions, we deal with S as a set of moving objects, ignoring the technical detail of its representation as a stream of tuples.

The evaluation of the first form of the *gpattern* operator is:

eval(S **gpattern**[id, α, d, n, "*exactly*"]) =

$$\begin{aligned}\{\{(I,V)\} | & V \subseteq S,\ |V| = n, I \in Interval, \textbf{length}(I) \geq d,\\ & \forall t \in I, \forall e \in V : (\alpha(e))(t),\\ & \forall I' \text{ such that } I' \in Interval, I \subset I' :\\ & \exists e \in V, \exists t \in I' \ :\ \neg(\alpha(e))(t)\}\end{aligned}$$

where $\textbf{length}(I) = I.t_2 - I.t_1$, and $(\alpha(e))(t) \in$ {false, true} is the evaluation of the time-dependent predicate α for the moving object e at the time instant t. The operator yields groups of exactly n moving objects. Therefore, every $\underline{mset}$ in the result contains only one unit (I, V). The last condition guarantees that only the longest duration patternoids are reported, to avoid an infinite number of results.

The definition allows a moving object to belong to several groups in the result. This is required to report all possible combinations of exactly n objects. This is helpful in expressing the *trend-setter* pattern, for instance. That is, a group of exactly k moving objects matches some patternoid description, followed by another group of at least j moving objects that matches the same description.

To define the semantics of the second form of the *gpattern* operator, we need to define some auxiliary operations:

$\underline{mbool}$	$\rightarrow \underline{bool}$	**always**	$\#(\ \_\ )$
$\underline{mbool}$	$\rightarrow \underline{bool}$	**sometimes**	$\#(\ \_\ )$
$\underline{mset} \times \underline{set}$	$\rightarrow \underline{mbool}$	$\subset, \subseteq$	$\_ \# \_$
$data \times \underline{mset}$	$\rightarrow \underline{mbool}$	$\in$	$\_ \# \_$
$mapping$	$\rightarrow \underline{range}(\underline{instant})$	**deftime**	$\#(\ \_\ )$
$\underline{mbool} \times \underline{bool}$	$\rightarrow \underline{mbool}$	**at**	$\_ \# \_$

where *always* yields true if its argument has the value true all over its definition time. On the other hand, *sometimes* yields true if its argument is ever true. The time-dependent versions of the set predicates $\subset, \subseteq$, and $\in$ yield true at the time instants/intervals during which their corresponding standard predicates hold.[1] The *deftime* operation yields the set of time intervals and/or instants during which a moving object is defined. Finally, the *at* operation restricts the definition time of a moving object to the time intervals and/or instants during which its value is equal to the second argument.

Given a *gpattern* operator in the form S **gpattern**[id, α, d, n, "*atleast*"], we first define the set M as:

$$
\begin{aligned}
M = \{X | X \in MSetPart, & \textbf{always}(X \subseteq S),\ \textbf{always}(|X| \geq n), \\
& (\forall e \text{ such that } \textbf{sometimes}(e \in X) : \\
& \quad P = \ \textbf{deftime}((e \in X)\ \textbf{at}\ \text{true}) \Rightarrow \\
& \quad (i)\ \forall I \in P : \textbf{length}(I) \geq d \\
& \quad (ii)\ \forall t \in P : (\alpha(e))(t))\}
\end{aligned}
$$

The set M contains all possible matches of the patternoid. A group of moving objects that matches this patternoid can change (i.e. objects may join and/or leave the group). The last four lines ensure that, each time an object joins the group, it stays for a duration of at least d. The set M might contain an infinite number of matches. To avoid this, we restrict it to the matches that are maximal in the number of moving objects, and maximal in their definition times. That is:

eval(S **gpattern**[id, α, d, n, "*atleast*"]) =

$$
\begin{aligned}
\{X | X \in M, & \\
& (\nexists Y \in M \ \text{ such that } \textbf{deftime}(X) \subset \textbf{deftime}(Y)), \\
& (\nexists Y \in M \ \text{ such that } (\textbf{deftime}(X) = \textbf{deftime}(Y) \wedge \textbf{sometimes}(X \subset Y)))\}
\end{aligned}
$$

The evaluation of the *gpattern* operator is illustrated in Algorithm 3. It uses the auxiliary functions shown in Algorithms 4 - 6. It also uses the time-dependent *union* operation for $\underline{mset}$s, which is defined as follows:

$$
(o_1 \cup o_2)(t) = \begin{cases} o_1(t) \cup o_2(t), & \text{if isdef}(o_1(t)) \wedge \text{isdef}(o_2(t)) \\ o_1(t), & \text{if isdef}(o_1(t)) \wedge \neg\text{isdef}(o_2(t)) \\ o_2(t), & \text{if}\neg\text{isdef}(o_1(t)) \wedge \text{isdef}(o_2(t)) \\ \text{undef} & \text{if}\neg\text{isdef}(o_1(t)) \wedge \neg\text{isdef}(o_2(t)) \end{cases}
$$

where o_1, o_2 are $\underline{mset}$ instances, t is a time instant, and *isdef* yields true iff its argument is defined.

Algorithm 3: gpattern(S: $\underline{stream}$(*tuple*), id: (*tuple* $\rightarrow$ $\underline{int}$), α: (*tuple* $\rightarrow$ $\underline{mbool}$), n: $\underline{int}$, d: $\underline{duration}$, q: enum{exactly, atleast}) $\rightarrow$ $\underline{stream}$($\underline{mset}$)

```
1  let R be a stream(mset), initially empty;
2  let Accumulator be an mset, initially empty;
3  foreach s: tuple in S do Accumulator = Accumulator ∪ mbool2mset(α(s), id(s));
4  let Changed := true;
5  while Changed do
6  |   Changed := DeleteUnitsBelowSize(Accumulator, n);
7  |   Changed := Changed ∨ DeleteElemsBelowDuration(Accumulator, d);
8  while Accumulator has more units do
9  |   let head := Head(Accumulator);
10 |   Accumulator := Rest(Accumulator);
11 |   if q = atleast then R.Add(head);
12 |   else R.Add( ExactSubsets(head, n, d));
13 return R;
```

Algorithm 4: mbool2mset(mb: $\underline{mbool}$, id: $\underline{int}$) $\rightarrow$ $\underline{mset}$

```
1  let ms be an mset, initially empty;
2  foreach (I, true): ubool in mb do ms.AddUnit(I, {id});
3  return ms;
```

Algorithm 3 accumulates the evaluations of the time-dependent predicate, for all the moving objects in the input stream, in the $\underline{mset}$ instance *Accumulator*. A moving object identifier appears in the *Accumulator* in the time intervals/instants during which it fulfills the time-dependent predicate. The size and duration thresholds n, d are then applied to the *Accumulator*. Finally, the last *while loop* iterates over the *Accumulator* to generate the result stream.

In every iteration, units are read till a definition gap is met (i.e. a time interval/instant during which the *Accumulator* is undefined). This is called the *head*, and it belongs to the set *MSetPart*. Therefore, *head* is already one result for the *gpattern* in the case of *atleast*. In the case of *exactly*, all the maximal duration subsets of *head* that have a cardinality of n are generated and added to the result stream.

7 The Crosspattern Operator

The *crosspattern* operator expresses a patternoid in terms of a time-dependent predicate evaluated for pairs of moving objects. Many patternoids can be expressed in this way (e.g. flock, convoy, convergence, etc.). Hence, it allows one to express patternoids on the spatiotemporal relationships between objects.

Formally, let S be a set of moving objects. Let α be a time-dependent predicate that can be applied to pairs of moving objects. We define the *pattern graph* as follows:

$$PG(S, \alpha) = \{(n_1, n_2, p) | n_1, n_2 \in S, p \in Range(Instant), \forall t \in p : (\alpha(n_1, n_2))(t)\}$$

That is, a *pattern graph* is a set of time-dependent edges in the form (n_1, n_2, p). An edge connects its two vertices whenever the time-dependent predicate α is fulfilled. The set of graph

[1] When the set operations $\subset$, $\subseteq$, and $\in$ are used in queries and typed at a keyboard, they are denoted as *isproppersubset*, *issubset*, and *in*, respectively.

Algorithm 5: DeleteUnitsBelowSize($Accumulator$: $\underline{mset}$, n: $\underline{int}$) $\rightarrow$ $\underline{bool}$

```
1 let Changed be a bool, initially false;
2 foreach (I, s): uset in Accumulator do
3     if s.CountElems() < n then
4         Accumulator.DeleteUnit((I, s));
5         Changed := true;
6 return Changed;
```

Algorithm 6: DeleteElemsBelowDuration($Accumulator$: $\underline{mset}$, d: $\underline{duration}$) $\rightarrow$ $\underline{bool}$

```
1 let Changed be a bool, initially false;
2 let s be a set, initialized by all the moving object identifiers in all the units of
  Accumulator;
3 foreach elem: int in s do
4     let p := deftime(elem ∈ Accumulator);
5     foreach I : interval in p do
6         if length(I) < d then
7             Accumulator.DeleteElemPart(elem, I);
8             Changed := true ;
9 return Changed;
```

vertices is not explicitly represented in the pattern graph. Rather, a vertex exists in the graph as long as at least one edge connects to it. Hence, the set of vertices is also time-dependent. The pattern graph is a kind of an analytical space, where patternoids can be searched for. The connectivity of the graph tells about the interaction between the moving objects based on the α function.

A temporal scan of a pattern graph yields a *fully dynamic graph* (i.e. a standard graph that undergoes a sequence of edge/node additions and/or deletions). There exist already, in the area of graph theory, algorithms for answering connectivity queries on *dynamic graphs* (e.g. finding connected components). These algorithms efficiently update the solution after a change happens to the graph, rather than re-evaluating the query from scratch. A review of such techniques can be found in [6]. We can safely assume that all types of connectivity queries that are supported for standard graphs, are also supported for dynamic graphs (e.g. Walk, Clique, Connected Component). That is, if we lack algorithms that efficiently search for certain subgraph types in a dynamic graph, the search will be done inefficiently (i.e. by re-evaluating the query after every change to the graph). The *crosspattern* operator is able to search for various subgraph types within the *pattern graph* (e.g. Walk, Clique, Knot) using such techniques.

Given a pattern graph $PG(S, \alpha)$ and an instant of time t, the temporal function $PG(S, \alpha)(t)$ yields the standard graph (N, E), where E is the set of edges in $PG(S, \alpha)$ that are defined at time t, and N is the set of nodes connected to at least one edge in E.

In the following definition, we use the $\underline{mset}$ type to represent the results of the crosspattern operator. The elements of the $\underline{mset}$ are the moving objects (i.e. the nodes of the pattern graph) that fulfill the patternoid. Note that such a representation ignores the graph edges in the results.

Let $U \subseteq S \times S$ be a set of candidate pairs[2] of elements from S and let $S' = \pi_1(U) \cup \pi_2(U)$,

[2]We use U as input to the crosspattern operator instead of S to be able to do some prefiltering. For example, interesting pairs of candidates from S may be those coming close to each other during their lifetime, and they

where for a set of tuples V, $\pi_i(V)$ denotes the projection on the i-th component. So S is reduced to S', the elements mentioned in U. Given an application of the *crosspattern* operator in the form U **crosspattern**[id_1, id_2, α, d, n, "clique"], we first define the set Q as:

$$\begin{aligned} Q = \{ & X \in MSetPart \mid \textbf{always}(X \subseteq S') \\ & \wedge (\forall (I, V) \in X, \forall t \in I : \\ & \quad V \text{ is a } \textit{maximal clique} \text{ in PG}(S', \alpha)(t) \wedge |V| \geq n) \\ & \wedge (\forall e \text{ such that } \textbf{sometimes}(e \in X) : \\ & \quad P = \textbf{deftime}((e \in X) \textbf{ at } \text{true}) \implies \forall I \in P) : \textbf{length}(I) \geq d\} \end{aligned}$$

An element $X \in Q$ is an $\underline{mset}$ instance representing one group of moving objects that match the patternoid. The *maximal clique* condition guarantees that the number of moving objects in a result is maximal. We mention the *clique* in this definition as an example. Clearly it can be replaced by other subgraph types. Now we define the evaluation of the *crosspattern* operator as:

eval(U **crosspattern**[id_1, id_2, α, d, n, "clique"]) =

$$\begin{aligned} \{X \in Q \mid & \nexists Y \in Q \text{ such that } (\textbf{deftime}(X) \subset \textbf{deftime}(Y) \wedge \\ & (\forall t \in \textbf{deftime}(X) : X(t) = Y(t)))\} \end{aligned}$$

where $X(t), Y(t)$ are the $\underline{set}$ values of X, Y at time t. This definition adds to the definition of Q the condition that the definition time of a result is maximal, thus avoiding an infinite number of results.

The evaluation algorithm of the *crosspattern* operator is not listed in this paper to keep the scope of the paper focused. It requires the introduction of specialized data structures for answering connectivity queries on the *pattern graph*. While the $\underline{mset}$ type was sufficient for the definitions above, it is not sufficient for the evaluation algorithm since it doesn't store edge information. We plan to present these data structures and the algorithm in a separate manuscript.

8 Examples

This section illustrates the expressive power of the proposed language by showing several query examples. We focus on expressing the patternoids in Table 1, as a kind of comparison with the other techniques.

Example 4 Find a flock of at least 20 gazelles moving within a circle of radius 30 m for a duration of 10 minutes.

This is a $(20, 30m, 10min)$ *flock*, similar to number 8 in Table 1. The difference is, that the notion of time here is continuous. The query looks as follows:

```
query Gazelles feed {a} Gazelles feed {b}
  symmjoin[.Id_a < ..Id_b]
  crosspattern[.Id_a, .Id_b,
    distance(.Trip_a, .Trip_b) < 30.0, ten_minutes, 20, "clique" )]
  transformstream consume;
```

can possibly be determined efficiently using indexes. Evaluating instead all pairs from S may be prohibitively expensive.

Example 5 Find a convoy of at least 20 gazelles moving within a circle of radius 30 m for a duration of 10 minutes.

The convoy patternoid (number 7 in Table 1) is a variant of the above flock patternoid. Unlike a flock, the members of a convoy may change. This can be expressed as follows:

```
query Gazelles feed {a} Gazelles feed {b}
  symmjoin[.Id_a < ..Id_b]
  crosspattern[.Id_a, .Id_b,
    distance(.Trip_a, .Trip_b) < 30.0,
    one_minute, 20, "clique" )]
  transformstream
  filter[inst(final(.elem)) - inst(initial(.elem))
    >= ten_minutes] consume;
```

That is, every object in the patternoid stays at least one minute, while the patternoid itself exists for at least ten minutes, as enforced by the *filter* operator. The effect of this is, that the objects of the patternoid may change.

Example 6 Find a leadership pattern that consists of: 3 gazelles heading in Northern direction, followed by a moving cluster of 20 gazelles.

A moving cluster is another variant of a flock. Every member is required to keep a small distance to *some* (rather than *all*) other members.

```
let follow = vec("baba", "bab.a", "baab");
query
  reportpattern[
    leader: Gazelles feed
      gpattern[.Id, mdirection(.Trip) between
        [45.0, 135.0], three_minutes, 3, "exactly"],
    cluster: Gazelles feed {g1} Gazelles feed {g2}
      symmjoin[.Id_g1 < ..Id_g2]
      crosspattern[.Id_g1, .Id_g2,
        distance(Trip_g1, Trip_g2) < 30.0,
        ten_minutes, 20, "cc" ];
    stconstraint(.cluster, .leader, follow)]
  extend[clusterMReg: fun(t: TUPLE) Gazelles feed
    mset2mreg(.Id, .Trip, attr(t, cluster))]
  extend[leaderFinSet: val(final(.leader)),
    clusterInitSet: val(initial(.cluster))]
  filter[always(mdirection(rough_center(.clusterMReg))
    between [45.0, 135.0]) and
    (.leaderFinSet issubset .clusterInitSet)]
  consume;
```

where *mdirection* computes the time-dependent angle, in degrees, of the moving point from the x-axis, and *rough_center* computes the center of a moving region as a moving point. The query finds a *leader* group, followed by a moving cluster, where both are heading between north-east and north-west. The moving cluster patternoid is expressed as a *connected component* (cc) in the pattern graph, rather than a *clique* in the case of the flock patternoid in Example 4.

Example 7 The encounter patternoid is a variant of the convergence patternoid, in Example 2. It is a convergence that ends with the moving objects meeting together. We express it as follows:

```
query Gazelles feed {a} Gazelles feed {b}
  symmjoin[.Id_a < ..Id_b]
  crosspattern[.Id_a, .Id_b,
    isdecreasing(distance(.Trip_a, .Trip_b)),
    20, ten_minutes, "clique" )]
  transformstream
  extend[convMReg: fun(t2: TUPLE) Gazelles feed
    mset2mreg(.Id, .Trip, attr(t2, .elem)]
  filter[area(val(final(.convMReg)))
    <= const_pi() * 30.0 * 30.0]
  consume;
```

that is, the gazelles at the end instant of the patternoid are required to be within a circular area with radius of 30 m.

9 Optimization

This section discusses the optimization of the group STP queries. We assume that an optimizer framework for moving object databases exists. The techniques that we propose in this section extend such an optimizer framework, so that it is able to generate optimized execution plans for the group STP queries. In order to make this discussion concrete, we describe the proposed optimization techniques in the context of the SECONDO optimizer [11]. These techniques are however generic, and it should be possible to apply them in other optimizer frameworks.

The SECONDO optimizer accepts queries in an SQL-like syntax, and generates optimal execution plans in the syntax that was used for the query examples in the preceding sections (i.e. the SECONDO executable language). We start by defining an SQL-like syntax for the proposed language operators. The user is supposed to use it in writing the group STP queries. We then define translation rules to map these SQL-like queries into efficient execution plans.

Since the three operators *reportpattern*, *gpattern*, and *crosspattern* yield streams, we define them in the SQL-like syntax as *table expressions* (i.e. expressions that compute tables). That is, they can be used in the FROM clause, for instance. The syntax is as follows:

```
table_expr ::= reportpattern( [ p_expr, p_expr [, p_expr ...] ],
                 [temporal_constraint [, temporal_constraint ...]] )
p_expr     ::= p_operator as alias
p_operator ::= gpattern( table_expr, id, α, d, n, q ) |
               crosspattern( table_expr, id1, id2, α, d, n, q )
```

The following query illustrates the SQL-like syntax for Example 6:

```
SELECT leader, cluster,
FROM (
  SELECT leader, cluster,
    val(final(leader)) as leaderFinSet,
    val(initial(cluster)) as clusterInitSet,
    mset2mreg(Gazelles, Id, Trip, cluster) as clusterMReg
  FROM
    reportpattern([
      gpattern(Gazelles, id, mdirection(Trip) between [45.0, 135.0],
        three minutes, 3, "exactly") as leader,
      crosspattern(
        ( SELECT * FROM Gazelles g1, Gazelles g2
          WHERE g1.Id < g2.Id ),
```

```
        g1.Id, g2.Id, distance(g1.Trip, g2.Trip)< 30,
        ten minutes, 20, "scc")] as cluster,
      [stconstraint(cluster, leader, follow)] ))
WHERE
  [always(mdirection(rough_center(clusterMReg)) between [45.0, 135.0]),
  leaderFinSet issubset clusterInitSet]
```

The straightforward *translation rule* of the *reportpattern* operator is as follows:

reportpattern([p_1 **as** a_1, ..., p_n **as** a_n], [tc_1, ..., tc_m])
$\rightarrow$ reportpattern[$a_1 : p_1$, ..., $a_n : p_n$; tc_1, ..., tc_m]

where a *translation rule* defines one way of generating the execution plan for an SQL-like expression. It has the structure:

SQL-like_expr $\rightarrow$ *Executable_expr* [:- *conditions*]

The optional conditions at the end of the rule specify the circumstances under which the rule is applicable. There may be several valid translations for the same expression. Optimizers apply different techniques for selecting the best translation (e.g. cost-based optimization). The following subsections illustrate the translation rules for the three operators *gpattern*, *crosspattern*, and *reportpattern*.

9.1 Optimizing the Gpattern Operator

The *gpattern* operator evaluates for every tuple in the input stream the time-dependent predicate. Obviously, a tuple whose evaluated $\underline{mbool}$ is always false cannot be part of a result. We would like to use indexes to remove such tuples from the input stream before evaluating the *gpattern* operator. The optimizer should be able to analyze the time-dependent predicates and translate them into index accesses if possible. In other words, we need to extend the optimizer with translation rules for the time-dependent predicates.

We assume that the underlying system has already translation rules for the standard (i.e. non-temporal) selection predicates. That is, given a query that consists of a table expression *in_stream*, and a standard selection predicate *f* that is applied to it, it is possible to invoke a function *optimize*(*in_stream*, *f*), which yields an optimal execution plan for this query. If the *in_stream* is a scan of a database relation, *optimize* might yield an index scan if relevant. If it is not possible to use indexes, it yields the straightforward plan in_stream **filter**$[f]$.

We define the translation rules of the time-dependent predicates on top of the translation rules of the standard predicates. A similar approach was successfully adopted in [17]. The idea is to map every time-dependent predicate p into a standard predicate f such that $f \Leftrightarrow$ **sometimes**(p). For example:

$\underline{mpoint} \times \underline{region} \quad \rightarrow \quad \underline{mbool} \qquad$ **inside** $\qquad$ _ # _

is mapped into the standard predicate:

$\underline{mpoint} \times \underline{region} \quad \rightarrow \quad \underline{bool} \qquad$ **passes** $\qquad$ _ # _

where *inside* is the time-dependent predicate illustrated in Figure 2, and *passes* is a standard predicate that yields true if the $\underline{mpoint}$ object ever passes through the $\underline{region}$ object. A list of such mappings can be found in [17].

This mapping allows for using the existing optimization framework of standard predicates, in order to optimize the time-dependent predicates, and accordingly the *gpattern* operator. That is, let *map*(p) denote the standard predicate mapping (as explained above) of the time-dependent predicate p. The translation rule of the *gpattern* operator is as follows:

gpattern(S, id, α, d, n, q)
$\rightarrow optimize(S, map(\alpha))$ gpattern[id, α, d, n, q]

The following example illustrates this translation rule. Suppose that we wish to express a group STP query called *drinking*, that reports groups of gazelles that gather to drink from some lake. The SQL-like query would be as follows:

```
SELECT mset2mreg(Gazelles, Id, Trip, elem) as drinking
FROM gpattern(
  Gazelles, Id, Trip inside buffer(theLake, 50),
  half_hour, 20, "atleast");
```

The *gpattern* operator in this query yields a relation that contains one $\underline{mset}$ attribute with the default name *elem*. The *buffer* operator creates a zone around the region *theLake* with an extent of 50 m. The query finds groups of gazelles that concurrently stay within a distance of 50 m from *theLake*, for at least half an hour. A possible execution plan that the optimizer generates for this query is:

```
query Gazelles_Trip_sptuni
  windowintersectsS[bbox(buffer(theLake, 50))]
  sort rdup Gazelles gettuples
  filter[.Trip passes buffer(theLake, 50)]
  gpattern[.Id, .Trip inside buffer(theLake, 50),
    half_hour, 20, "atleast")
  transformstream
  extend[drinking: fun(t: TUPLE) Gazelles feed
    mset2mreg(.Id, .Trip, attr(t, elem))
  consume;
```

In order for the optimizer to generate this execution plan, it first maps α (i.e. `Trip inside buffer(theLake, 50)`) into $map(\alpha)$ (i.e. `Trip passes buffer(theLake, 50)`). The `Gazelles_Trip_sptuni` in the execution plan is a spatial R-tree index on the units of the *Trip* attribute that we assume to exist in the database. The $optimize(S, map(\alpha))$ generates the index access that is illustrated in the execution plan, instead of simply scanning the *Gazelles* relation. It produces a window query to the R-tree index using the bounding box of the buffer region around the lake, and refines the result using the *passes* standard predicate. Thus, only the tuples that sometimes fulfill the time-dependent predicate *inside* are passed to the *gpattern* operator.

9.2 Optimizing the Crosspattern Operator

The optimization of the *crosspattern* operator is similar to that of the *gpattern* operator. The only difference is that the time-dependent predicate in the *crosspattern* operator is applied to pairs of moving objects. So, it is handled as a join predicate rather than a selection predicate as in the *gpattern* operator. Nevertheless, the strategy is the same.

The time-dependent predicate p in the *crosspattern* operator is mapped into a standard join predicate by $map(p)$. We assume that the optimizer defines a function $optimize(S_1, S_2, f)$ that accepts two table expressions S_1, S_2 and a standard join predicate f and yields an optimized execution plan. The translation rule hence is as follows:

crosspattern($S, id_1, id_2, \alpha, d, n, q$)
$\rightarrow$ *optimize*($S, S, map(\alpha)$) crosspattern[$id_1, id_2, \alpha, d, n, q$]

9.3 Optimizing the Reportpattern Operator

As described in Section 5, the *reportpattern* operator is evaluated incrementally by solving the CSP_{k-1} first, then extending it to CSP_k. During the evaluation, the algorithm knows temporal information from the patternoid operators evaluated so far, and from the temporal constraints. This temporal information can be used to restrict the definition time of the input trajectories before the evaluation of the remaining patternoid operators. Consider, for example, a *reportpattern* operator containing two patternoid operators p_1, p_2, and a temporal constraint *stconstraint*(p_1, p_2, *vec*("aabb")). If we know, after evaluating p_1, that the earliest ending group in its result ends at time t_1, a supported assignment of p_2 can only start after t_1. We can safely restrict the definition time of the input trajectories to $t > t_1$ before evaluating p_2. Thus, p_2 receives shorter trajectories, and its evaluation time decreases.

This would require a change to Algorithm 1 of the *reportpattern* operator. Line 5 of the algorithm needs to be replaced by two steps: (1) given the set of temporal constraints, the set of supported assignments SA, and the next patternoid operator to be evaluated p_i, the first step is to perform a *temporal reasoning* to compute the time periods on which a solution of p_i can be consistent, (2) to tell the patternoid operator p_i to use these computed time periods to restrict the definition times of the input trajectories before evaluating its time-dependent predicate.

It is possible to transform the temporal constraints within the *reportpattern* operator, so that the temporal reasoning is performed using the *Basic Point Algebra* (BPA) [19]. The BPA performs reasoning over time points, and supports the four point relations $\{<, >, =, ?\}$, where $?$ is the universal constraint (i.e. can be replaced by any of the other three relations). In the following, we illustrate this transformation.

In Section 5, we have defined the set of temporal relationships IR. An element $ir \in IR$ can be decomposed into three time point relations/constraints between four time point variables. That is, given two time intervals i_1 and i_2, ir defines a total order $(t_1 \odot_1 t_2 \odot_2 t_3 \odot_3 t_4)$, where $\{t_1, t_2, t_3, t_4\} = \{i_1.t_1, i_1.t_2, i_2.t_1, i_2.t_2\}$, and $\odot_i \in \{<, =\}$.

It is, hence, possible to represent the set of temporal constraints within a given *reportpattern* operator as a BPA constraint network. Given a *reportpattern* operator with n patternoid operators and m temporal constraints each consisting of at most k terms, the corresponding BPA constraint network has $2n$ nodes and a maximum of $3mk$ constraints. The $2n$ nodes represent start and end time points of the n patternoids. The upper bound of $3mk$ constraints exists because each of the mk constraint terms is transformed into 3 BPA constraints. Note that k is bounded by 26, the size of the set IR, but in practice is a small number, as can be seen in the examples.

Checking the consistency of the BPA constraint network, and computing the closure (i.e. deriving the temporal relation between every pair of nodes) is performed in $O(n^3)$. The algorithm was first proposed by Allen [2] for his interval algebra, and was proven by Vilan et al. [19] to work for the BPA as well. It contains two interface functions: *Add*, and *Close*. The network is constructed by calling *Add* for every point constraint. The *Close* is called afterwards to compute the consistency, and the closure. A BPA network is either inconsistent, or it must induce a strict partial order [18].

The modified algorithm for evaluating *reportpattern* is illustrated in Algorithm 7. The function *ConstructBPANetwork* in Line 3 constructs the BPA constraint network and computes the closure using Allen's algorithm as described above. In every iteration, the consistent periods for the selected patternoid operator are computed, Lines 7 – 11. In the first iteration, the *sa_periods* object is set to the maximal time interval $[0, \infty]$, because there is no temporal information available yet to restrict the trajectories. Starting from the second iteration, the consistent periods of the selected patternoid operator p_i are computed w.r.t. every $sa \in SA$. The

union of all these time periods is the result of the temporal reasoning.

Algorithm 7: *reportpattern*

Input: *P*: $(\underline{ident}, \underline{stream}(\underline{mset}))^+$ - a set of patternoid operators, *C*: $(tuple \rightarrow \underline{bool})^+$ - a set of temporal constraints

Output: *R*: $\underline{stream}(\underline{tuple}(<(\underline{ident}, \underline{mset})^+>))$

```
1  let SA: list(list(mset)) be a list of supported assignments, initially empty;
2  let Agenda := P;
3  let BPANetwork := ConstructBPANetwork(P, C);
4  if BPANetwork is not consistent then return an empty stream;
5  while Agenda is not empty do
6      i := pick p_i from Agenda;
7      let sa_periods: range(instant) be initially empty;
8      if SA is empty then sa_periods := [0, ∞];
9      foreach sa in SA do
10         let cp := GetConsistentPeriods(BPANetwork, sa, p_i);
11         sa_periods := sa_periods ∪ cp;
12     SetArgument(p_i, sa_periods);
13     let d:= evaluate p_i.second;
14     extend(SA, d, i, C);
15     if SA is empty then
16         return an empty stream;
17 let R := construct the result stream from SA;
18 return R;
```

It remains now to use the computed *sa_periods* to restrict the trajectories' definition time. To do this, we need to redefine the patternoid operators *gpattern* and *crosspattern* as functions that accept an argument of type $\underline{range}(\underline{instant})$. That is, a patternoid operator becomes a function with the signature:

$$\underline{periods} \rightarrow \underline{stream}(\underline{mset})$$

where $\underline{periods}$ denotes $\underline{range}(\underline{instant})$. Accordingly the signature of the *reportpattern* operator in Table 3 is changed into:

$$(\underline{ident} \times (\underline{periods} \rightarrow \underline{stream}(\underline{mset})))^+ \times (tuple \rightarrow \underline{bool})^+ \rightarrow \underline{stream}(\underline{tuple}(< (\underline{ident}, \underline{mset})^+ >)) \qquad \textbf{reportpattern} \quad \#[\_; \_]$$

This change in the signature allows the *reportpattern* operator to pass the computed *sa_periods* to the patternoid operators at run-time, as illustrated in line 12 of Algorithm 7.

At the same time, we wish to keep the signature of the operators *gpattern* and *reportpattern* unchanged, so that it is possible to use them outside the *reportpattern* operator, as in Examples 1, 2, 4, 5, and 7. Therefore, we keep the *gpattern* and *reportpattern* unchanged, and define similar operators *gpattern2* and *reportpattern2* that accept the additional $\underline{periods}$ argument. In the SQL-like syntax, the user does not need to pay attention to this syntactical difference. He/she will always be using the *gpattern* and the *crosspattern* operators in the queries. In the execution plan, they will be translated into *gpattern2* and *crosspattern2* if they happen to be inside the *reportpattern* operator. The translation rule of the *reportpattern* operator instructs the optimizer to do this. We omit it here, because it is trivial.

Before illustrating the translation rules for the *gpattern2* and the *crosspattern2* operators, we show in the following a part of the execution plan that the optimizer should produce for the *reportpattern* operator in the SQL-like query in Section 9.

```
query reportpattern[
  leader: fun(p: periods) Gazelles feed
    gpattern2[.Id, mdirection(.Trip atperiods p)
      between [45.0, 135.0],
      three_minutes, 3, "exactly"],
...
```

Notice the function header *fun(p: periods)* that precedes the *gpattern2* operator. The function argument *p* is set by the *reportpattern* operator during execution. Notice also how this argument is used to restrict the gazelle trajectories in the part *.Trip atperiods p*. The operator *atperiods* restricts the definition time of a moving object to a given $\underline{periods}$ value.[3]

The translation rules for the operators *gpattern2* and *crosspattern2* need to parse the time-dependent predicate, and apply the *atperiods* operator to the moving object arguments coming from the input stream. The translation rules for the *gpattern2* operator would look like the following:

```
gpattern(S, Id, α, D, N, Q)
->  fun(Name: periods) optimize(S, map(α2))
      gpattern2[Id, α2, D, N, Q] :-
insideReportpattern,
generateRandomVariableName(Name),
isArgument(attr(Attr, S), α),
isMapping(attr(Attr, S)),
concat_atom([attr(Attr, S), 'atperiods', Name],
  '', Attr1),
replaceArgument(attr(Attr, S), α, Attr1, α2), !.

gpattern(S, Id, α, D, N, Q)
->  fun(Name: periods) optimize(S, map(α))
      gpattern2[Id, α, D, N, Q] :-
insideReportpattern,
generateRandomVariableName(Name).
```

where the conditions under which the rules apply are written in a Prolog syntax. The two rules make sure in the beginning that this translation happens inside the *reportpattern* operator, hence the SQL-like *gpattern* is translated into the executable *gpattern2*. The first rule adds the time-dependent predicate α to the execution plan modified as $\alpha 2$. This is done under the conditions that there are some arguments of α that are attributes in the tuple stream S, and that they are of the kind MAPPING. In such a case, the *atperiods* operator is applied to these arguments in the execution plan. If any of the two conditions fail, the second rule adds α unchanged to the execution plan. Similar translation rules are added also for the *crosspattern2* operator.

10 Conclusions

In this paper, we have proposed a language for group spatiotemporal pattern (STP) queries. The language is both expressive and extensible. The query examples in the paper show that arbitrarily complex group STP queries can be expressed. Essentially patterns are composed of patternoids, and patternoids are expressed on top of time-dependent predicates. Each of these

[3] Actually, the optimization method uses a time interval found for the first patternoid to restrict the time intervals that need to be considered for the second and further patternoids. Hence in this example, the restriction would be really effective for the second, the *crosspattern* operator. Nevertheless we explain the mechanism in terms of the *gpattern* operator.

three layers is both expressive and extensible. The overall expressive power of the language is a kind of multiplication of their expressive powers.

In this paper, we have defined two patternoid operators. They are able to express patternoids in terms of the *independent movement*, and the *dual interaction* between moving objects. An opportunity for future work is to propose a patternoid operator for *team interactions*. Such patternoids can occur in soccer games, for instance (e.g. the offside trap). Mainly they describe groups of moving objects, where every individual shows some individual movement pattern, and these patterns relate together and show some group STP.

The representation of the pattern results as $\underline{mset}$s allows for nesting group STP queries into more complex queries. It is possible, for instance, to build individual STP queries on the results of a group STP query. That is, one can create moving region representations of the $\underline{mset}$ results and further process these moving regions. One can also build group STP queries on the results of other group STP queries (e.g. find a group of at least 5 animal flocks that meet around a lake).

We have also proposed methods of optimization that are extensible. They do not require specialized index structures. Rather, they are built on top of the existing optimization framework.

The language design that we proposed in this paper takes into consideration the integration with a moving object DBMS. We are currently implementing it in the context of the SECONDO system. It is an extensible DBMS framework where a large part of the moving objects model that we assume is implemented. We also intend to elaborate more on the optimization part and work it out in more detail.

References

[1] SECONDO web site. http://dna.fernuni-hagen.de/secondo.html/.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[3] N. Andrienko and G. Andrienko. Designing visual analytics methods for massive collections of movement data. *Cartographica*, 42(2):117–138, 2007.

[4] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. *Comput. Geom. Theory Appl.*, 41(3):111–125, 2008.

[5] S. Dodge, R. Weibel, and A.-K. Lautenschütz. Towards a taxonomy of movement patterns. *Information Visualization*, 7(3):240–252, 2008.

[6] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

[7] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 319–330, New York, NY, USA, 2000. ACM.

[8] J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal data sets. In *GIS '04: Proceedings of the 12th annual ACM International Workshop on Geographic Information Systems*, pages 250–257, New York, NY, USA, 2004. ACM.

[9] R. H. Güting. Second-order signature: a tool for specifying data models, query processing, and optimization. *SIGMOD Rec.*, 22(2):277–286, 1993.

[10] R. H. Güting, V. Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle. SECONDO: An extensible DBMS platform for research prototyping and teaching. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 1115–1116, Washington, DC, USA, 2005. IEEE Computer Society.

[11] R. H. Güting, T. Behr, V. Almeida, Z. Ding, F. Hoffmann, and M. Spiekermann. SECONDO: An extensible DBMS architecture and prototype. Technical Report Informatik-Report 313, FernUniversität Hagen, March 2004.

[12] 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.

[13] H. Jeung, H. T. Shen, and X. Zhou. Convoy queries in spatio-temporal databases. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1457–1459, Washington, DC, USA, 2008. IEEE Computer Society.

[14] P. Laube, S. Imfeld, and R. Weibel. Discovering relative motion patterns in groups of moving point objects. *International Journal of Geographical Information Science*, 19(6):639–668, 2005.

[15] P. Laube, M. v. Kreveld, and S. Imfeld. Finding REMO - detecting relative motion patterns in geospatial lifelines. In *Developments in Spatial Data Handling: Proceedings of the 11th International Symposium on Spatial Data Handling*, pages 201–215, Berlin Heidelberg, 2004. Springer.

[16] Z. Li, M. Ji, J.-G. Lee, L.-A. Tang, Y. Yu, J. Han, and R. Kays. MoveMine: mining moving object databases. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 1203–1206, New York, NY, USA, 2010. ACM.

[17] M. A. Sakr and R. H. Güting. Spatiotemporal pattern queries. *GeoInformatica, online first*, 2010. DOI 10.1007/s10707-010-0114-3.

[18] E. Schwalb and L. Vila. Temporal constraints: A survey. *Constraints*, 3(2/3):129–149, 1998.

[19] M. Vilain, H. Kautz, and P. van Beek. Constraint propagation algorithms for temporal reasoning: a revised report. In D. S. Weld and J. d. Kleer, editors, *Readings in qualitative reasoning about physical systems*, pages 373–381, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

Verzeichnis der zuletzt erschienenen Informatik-Berichte

[341] Saatz, I.: Unterstützung des didaktisch-methodischen Designs durch einen Softwareassistenten im e-Learning

[342] Hönig, C. U.:
Optimales Task-Graph-Scheduling für homogene und heterogene Zielsysteme

[343] Güting, R. H.:
Operator-Based Query Progress Estimation

[344] Behr, T., Güting, R. H.:
User Defined Topological Predicates in Database Systems

[345] vor der Brück, T.; Helbig, H.; Leveling, J.:
The Readability Checker Delite Technical Report

[346] vor der Brück, T.:
Application of Machine Learning Algorithms for Automatic Knowledge Acquisition and Readability Analysis Technical Report

[347] Fechner, B.:
Dynamische Fehlererkennungs- und –behebungsmechanismen für verlässliche Mikroprozessoren

[348] Brattka, V., Dillhage, R., Grubba, T., Klutsch, A.:
CCA 2008 - Fifth International Conference on Computability and Complexity in Analysis

[349] Osterloh, A.:
A Lower Bound for Oblivious Dimensional Routing

[350] Osterloh, A., Keller, J.:
Das GCA-Modell im Vergleich zum PRAM-Modell

[351] Fechner, B.:
GPUs for Dependability

[352] Güting, R. H., Behr, T., Xu, J.:
Efficient *k*-Nearest Neighbor Search on Moving Object Trajectories

[353] Bauer, A., Dillhage, R., Hertling, P., Ko K.I., Rettinger, R.:
CCA 2009 Sixth International Conference on Computability and Complexity in Analysis

[354] Beierle, C., Kern-Isberner, G.
Relational Approaches to Knowledge Representation and Learning

[355] Sakr, M.A., Güting, R.H.
Spatiotemporal Pattern Queries

[356] Güting, R. H., Behr, T., Düntgen, C.: SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations

[357] Düntgen, C., Behr, T., Güting, R.H.: Assessing Representations for Moving Object Histories