Spatiotemporal Pattern Queries

Mahmoud Attia Sakr
Database Systems for New Applications
FernUniversität in Hagen, 58084 Hagen, Germany
Faculty of Computer and Information Sciences
University of Ain Shams, Cairo, Egypt
mahmoud.sakr@fernuni-hagen.de

Supervised By
Ralf Hartmut Güting
Database Systems for New Applications
FernUniversität in Hagen, 58084 Hagen, Germany
rhg@fernuni-hagen.de

ABSTRACT

Capturing moving objects data is now possible and becoming cheaper with the advances in the positioning and sensor technologies. The increasing amount of such data and the various fields of applications call for intensive research work for building a spatiotemporal DBMS. This involves several aspects such as modeling the moving objects data, designing query methods, query optimization, ...etc. This PhD project goes in this direction. Our goal is to design and implement a language for the *Spatiotemporal Pattern Queries* (STP queries). This includes the design of an expressive query language, the integration with the query optimizers for efficient evaluation, and the implementation within the context of a spatiotemporal DBMS. The STPs can be either individual or group according to the number of objects composing the pattern. Our work covers both types. We have already completed a solution for the individual STP queries. Currently we are working in group STP queries.

1. INTRODUCTION

This PhD project is part of the continuous work to represent and query moving objects [8] [7] [5]. It aims at supporting the complex class of queries called spatiotemporal pattern (STP) queries. Whereas the standard spatiotemporal queries selects the moving objects based on spatiotemporal predicates, STP queries selects them based on temporal arrangements of such predicates. A *spatiotemporal pattern* can be either individual or group according to the number of the objects that are involved in the pattern. An example for individual STP query would be: *find all trains that encountered a delay more than half an hour after passing through a snow storm*. The example shows an STP that is a sequence of two spatiotemporal predicates. The pattern can be completely evaluated for every *train*, hence the name *individual STP*. An individual STP query, hence, reports the moving objects that fulfill a set of spatiotemporal predicates in a certain a temporal arrangement.

VLDB 2010 PhD Workshop, September 13, 2010, Singapore

On the other hand, a *group STP query* reports the patterns that involve a collective movement of several moving objects. An example for group STP queries would be: *find a traffic pattern where cars comming from side roads aggregate together in a main street resulting in high traffic*. An answer is a group of cars that together fulfill the pattern.

The STP queries are required for many application domains such as the behavioral study of animals, traffic monitoring, crime investigation, ...etc. The topic has attracted many researchers as we discuss in Section 2. Most of the related work goes in the direction of providing algorithms for the efficient evaluation of specific STPs. With the exception of the works in [6] and [10], there are no proposals for a language for STP queries up to our knowledge. These two languages have limitations that we discuss in detail.

The first objective of this PhD project is to support both the individual and the group STP queries within the context of the moving objects model in [8] [7] [5]. The intended outcome is a language that can express and evaluate the two kinds. The language is required to cover a wide range of patterns. More specific, it should neither be restricted to certain types of moving objects, nor to certain kinds of spatiotemporal operations. Moving objects can be moving geometries (e.g. moving points, moving regions, and moving lines), moving scalars (e.g. moving integers, moving strings, moving booleans, ...etc) or moving collections (e.g. moving sets, moving graphs, ...etc). The set of spatiotemporal operations that can be applied to these moving types is large and extensible. The language for STP queries is therefore required to accept the newly added operations beforehand.

The second objective is to extend the query optimizer to support the STP queries. On the one hand, the language must always be able to compute the valid results. Whereas on the other hand, the optimizer should be able to suggest the use of available indexes when possible. Obviously this is mandatory for an efficient execution of STP queries on large moving objects databases. The optimizer extension of STP queries must allow for extending the set of moving objects types and/or the set of spatiotemporal operations.

The third objective is to implement the language and the optimizer extension within SECONDO [1] where a big part of the model in [8] [7] [5] is already realized. SECONDO is an open source extensible DBMS platform. It is suitable for experimenting with new data types and operations.

We have completed a solution for individual STP queries including the optimizer extension and the implementation in SECONDO. The results were demonstrated in [12]. The approach is available as a technical report in [11], and is

submitted as a journal manuscript. The complete implementation is available as an open source SECONDO Plugin on the SECONDO web site [1] along with a user manual and an application example.

Currently we are finalizing the formal design of the language of the group STP queries. We have also implemented parts of the design. The results so far are promising. We didn't yet start the design of the optimizer extension.

The rest of this paper is organized as follows: Section 2 presents the closely related work. Section 3 briefly explains the moving objects model that we assume in this PhD project. Our work is presented in Sections 4 and 5. Section 4 illustrates the solution of the individual STP queries. Section 5 skitches our design for group STP queries. Finally Section 6 concludes the paper.

2. RELATED WORK

In the literature, the two problems of individual STP queries and group STP queries are handled separately. The first is handled within the database literature while the second is handled within the data mining literature. There exist few proposals related to the individual STP queries. We describe here the two most relevant approaches [9] and [6]. A longer list of related work can be found in [11].

The work in [9] proposes an index structure and efficient algorithms to evaluate the individual STP queries. It is however restricted to the *moving*(*point*) data type and to the *spatial* and *neighborhood* predicates.

The work in [6] formally defines the class of predicates called *spatiotemporal developments*. They describe the change in the topological relationship between two moving objects with time. For example, the spatiotemporal development *p Crosses r*, where *p* is a *moving*(*point*) and *r* is a *moving*(*region*), is defined as:

```
Crosses= Disjoint meet Inside meet Disjoint
```

This language is limited because it can only describe the changes in the topological relationship between pairs of moving objects. An individual STP query would more generally describe the interaction of the moving object with many other objects in the spatiotemporal space, and would involve many kinds of predicates.

The problem of group STP queries was visited more often in the literature. One would identify three main approaches for the solutions. The first uses visual analytics and information visualization to help the end users better explore the trajectory data and find the patterns (e.g. the work by Andrienko and Andrienko [3]). The second would propose specialized algorithms that are able to report certain group STPs (e.g. the work to report flock patterns in [4]). The third is the language approach, that is defining a query language capable of expressing such queries. We follow this approach in this project. The only related example that we are informed about is the work by Laube [10].

Laube [10] proposed the REMO (RElative MOtion) matrix model. It is a two dimensional matrix, where the rows represent the moving objects, and the columns represent a series of time instants (i.e. a discrete representation of time). The entries of the matrix are some predefined motion attributes that are computed for the moving objects (e.g. object speed, azimuth, acceleration, ...etc). The REMO matrix hence is a two-dimensional string, where two dimensional patterns can be searched for. The patterns are expressed by means of a regular language.

The REMO model is conceptually clean and highly expressive. We see, however, the following shortcomings in it: **(1)** The size of the REMO matrix is proportional to the database size. One would even need to maintain several matrixes for several motion attributes; **(2)** The REMO matrix does not inherently support the spatial proximity constraints between the moving objects. Such constraints are additionally applied to the results. For example, according to the REMO model a *flock* pattern is a *concurrency* pattern plus spatial proximity constraints. This would require that the flock members fulfill the concurrency pattern (i.e. concurrently share similar values for the motion attribute). Such a definition is restrictive. In reality, the flock members may not share common motion attributes for the whole duration (e.g. some cows in a flock of cows may move around, close to their flocks, with different speed and/or azimuth); **(3)** The model can not 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 expressed based on the mutual relationships between the moving objects (e.g. north of, closer than, intersects, ...etc.) can not be expressed; **(4)** The REMO matrix handles the time discretely. It does not directly support continuous trajectories. They have to be sampled first, which is known to incur inaccuracies in the data representation.

Beside the above limitations, none of the related work addresses the issues of query optimization and system integration, which are among the objectives of this PhD project.

3. THE MOVING OBJECTS DATA MODEL

This work is based on the *abstract data types* (ADT) model for moving objects in [8] [7] [5]. The model defines the *moving* type constructor. It constructs the moving counterpart of every static data type. Moving geometries for example are represented using three abstractions: *moving*(*point*), *moving*(*region*), and *moving*(*line*). In the abstract model [8], moving objects are modeled as temporal functions that map time to geometry or value. For example, a *moving*(*point*) is modeled as a curve in the 3D space (i.e. time to the 2D space).

In [7] a discrete data model implementing the abstract model is defined. In the discrete model, moving types are represented by the *sliced representation* as *units*.

DEFINITION 1. A data type *moving*(α) is a temporally ordered sequence of units. Every unit is a pair (I, *Instant* $\rightarrow \alpha$). The semantic of a unit is that at any time instant during the interval I, the value of the instance can be calculated from the temporal function *Instant* $\rightarrow \alpha$. Unit intervals are not allowed to overlap, yet gaps are possible (i.e. periods during which the value of the object is undefined). □

The moving data types are denoted by appending m to the standard type (e.g. *mpoint*). Similarly, the unit types are denoted by appending u. The *mpoint*, for example, is modeled in the discrete model as a temporally ordered list of *upoint*s, each of which consists of a time interval and a line function. The coordinates of the *mpoint* at any time instance within the interval are obtained by evaluating the line function. The *moving* type constructor is similarly applied to the scalar data types (e.g. *real*, *bool*), and the collection datatypes (e.g. *set*, *graph*).

Similar to the *moving* type constructor for the data types, the *temporal lifting* operation [8] is applied to all the standard operations to obtain their *lifted* counterparts. A lifted operation is obtained by allowing one or more of the arguments of a standard operation to be of a *moving* data type. For example, a static predicate and its corresponding lifted predicate are defined as follows;

DEFINITION 2. A *static predicate* is a function with the signature

$$P_1 \times .... \times P_n \rightarrow bool$$

where P_i is any static data type (e.g. *integer*, *point*, *region*). □

Example: Berlin *inside* Germany.

DEFINITION 3. A *lifted predicate* is a function with the signature

$$P_1 \times .... \times P_k \times \uparrow P_{k+1} \times ... \times \uparrow P_n \rightarrow \uparrow bool$$

where $\uparrow$ is the *moving* type constructor. □

Example: Train_RE103 *inside* Berlin.

Since some of the arguments of the lifted predicate are *moving*, the return type is an *mbool*. Our language design builds on the concept of lifted predicates, hence we can easily leverage a considerable part of the available infrastructure.

4. INDIVIDUAL STP QUERIES

This section describes the first part of the PhD project. This part is already completed, that is the language design, the optimizer integration, and the implementation within SECONDO are all finished.

4.1 The STP Predicate

An individual STP query is a query that contains one or more individual STP predicates (STPP for short). An STPP describes the pattern as a set of spatiotemporal predicates that fulfill in a certain temporal arrangement (e.g. a sequence). To motivate the idea of our design, consider the following query:

Example: Find the snow storms that passed over the cities Berlin then Dresden in order, while the speed of the storm is at least 80 km/h.

The query describes an individual STP consisting of three spatiotemporal predicates: *storm intersects Berlin*, *storm intersects Dresden*, and *speed of storm* $\geq$ *80 km/h*. The predicates are required to happen in the order that the second is after the first, and the third happens during the first and the second.

We propose a two step modular language design of the STPP. The first step is to define a special kind of predicates on moving objects that report the time intervals, during which they are fulfilled. The second step is to check the temporal arrangement of the predicate fulfillments.

Fortunately, the *lifted predicates* in Definition 3 do exactly what we need in the first step. They are a general and powerful class of predicates, as they are simply the time dependent version of arbitrary static predicates. They are not restricted to certain data types of arguments nor to certain types of operations.

In the second step, the STPP processes the *mbool* results of the lifted predicates and checks the temporal arrangement. It is modeled as a *constraint satisfaction problem* (CSP). All the constraints are binary temporal constraints, each of which states the temporal arrangement between two of the *lifted predicates*. In Definitions 4, 5, and 6 we define the parts that compose the STPP. The STPP itself is defined in Definition 7.

DEFINITION 4. A *temporal constraint* is a binary constraint that accepts two *mbool* parameters and checks a certain temporal arrangement between the pairs of their units. It has the signature

$$mbool \times mbool \times stvector \rightarrow bool$$

□

where the *stvector* argument describes the temporal arrangement between the two *mbool* arguments. It represents a vector temporal connector, as defined below in Definition 6. Syntactically the operator $stconstraint(\_, \_, \_)$ is used to express a temporal constraint.

Assume the expression $stconstraint(P, Q, V)$ where P and Q are lifted predicates each returning an *mbool* value, and V is a vector temporal connector. Let the set of time intervals of the units during which P is true be called P^{true} and similarly Q^{true} for Q. The temporal constraint is evaluated by calculating the Cartesian product of P^{true} and Q^{true}, then applying the vector temporal connector V on every pair of time intervals. It is fulfilled if one or more pairs fulfill V and we call such a pair a *supported assignment*.

Temporal connectors can be simple or vectors as shown in Definitions 5, 6.

DEFINITION 5. *Simple temporal connectors* are temporal connectors that enforce only one interval relationship. The set of simple temporal connectors is inspired from the 13 Allen's operators [2] with the addition that the intervals may degenerate into time instants. Hence 26 simple temporal connectors are possible. □

We define a language for writing the simple temporal connectors. The letters *aa* denote the begin and end time instants of the first argument. Similarly *bb* are the begin and end of the second argument. The order of letters describes the constraint, that is, a sequence *ab* means $a < b$. The dot symbol denotes the equality constraint, hence, the sequence *a.b* means $a = b$. A constraint must contain all the four letters. The complete list of the simple temporal connectors can be found in [11]. A temporal connector can alternatively be written as a vector of simple temporal connectors.

DEFINITION 6. A *vector temporal connector* is a set of simple temporal connectors. Vectors are interpreted as the disjunction of their constituent simple temporal connectors. Hence 2^{26} vector temporal connectors are possible. □

In the following we use the operator *vec* as a tool for constructing these vectors. For instance, the vector (`vec(aabb, abab, aa.bb)`), is a temporal connector that is fulfilled if any of its three components is fulfilled. It expresses the temporal arrangement that the interval *aa* starts before the start of the interval *bb*, and ends before the end of the interval *bb*. Now we define the spatiotemporal pattern predicate.

DEFINITION 7. A *spatiotemporal pattern predicate* (STPP) is a triple $\langle t, L, C \rangle$ where t is a tuple containing at least one

moving object, L is a set of lifted predicates that apply to the moving object in t and C is a set of temporal constraints. The predicate is fulfilled, if and only if the evaluations of the lifted predicates in L fulfill all the temporal constraints in C. In SQL, the operator *pattern* expresses the STPP. □

Example: Assuming the schema:
SnowStorm[id: *int*, storm: *mregion*]
Berlin: *region*, Dresden: *region*
The *snow storm* example in Section 4.1 may be written as follows:

```
SELECT id, storm
FROM SnowStorm
WHERE
  pattern([ storm intersects Berlin as onBerlin,
      storm intersects Dresden as onDresden,
      speed(center(storm)) >= 80.0 as highSpeedB,
      speed(center(storm)) >= 80.0 as highSpeedD],
  [stconstraint(onBerlin, onDresden,  vec(aabb)),
   stconstraint(highSpeedB, onBerlin, vec(abba, abb.a)),
   stconstraint(highSpeedD, onDresden, vec(abba, abb.a)])
```

Note that syntax of the STPP assigns *aliases* for the lifted predicates (using the *as* operator), so that they can be referenced in the temporal constraints. This is analogous to the aliases given to attributes and tables in the standard SQL.

In the query, *intersects* is a lifted predicate that is *true* whenever the spatial extent of the *mregion* argument intersects the *region* argument. The *center* operator accepts an *mregion* and yields an *mpoint* representing the region's center. The *speed* operator yields an *mreal* representing the moving speed of the *mpoint*. The *mreal* value is then compared to the value *80.0* using the lifted comparison predicate yielding an *mbool*. The lifted predicate (*speed*(.) >= .) is written twice within the *pattern* predicate with two different aliases *highSpeedB*, and *highSpeedD*. This is to make sure that the last two temporal constraints are independent. That is, the storm has to be moving fast on Berlin, and on Dresden but not necessarily in between.

This model allows for arbitrarily complex STP queries[1]. The naive evaluation of the STPP solves the CSP iteratively for every tuple. In the following subsection we describe how to integrate the STPP with the optimizer for efficient evaluation.

4.2 Optimizing the STPP

In the proposed design of the STPP, the moving objects data is processed by the lifted predicates in the first step. The second step processes the *mbool* results of the lifted predicates. In order to optimize the STPP, one would need a general technique for optimizing the lifted predicates.

Our idea is that, during the query rewriting phase of the optimization, an extra standard predicate (i.e. a predicate returning a *bool*) is added to the query for each of the lifted predicates in the STPP. The standard predicate is chosen according to the lifted predicate, so that the fulfillment of the standard predicate implies that the lifted predicate is fulfilled at least once. This is a necessary but not sufficient condition for the fulfillment of the STPP. In the following phases of optimization, the optimizer will be able to suggest optimized execution plans involving index accesses based on the extra standard predicates.

[1]In [11], we describe an extended version of the STPP that is more expressive. It is omitted here due to lack of space.

Table 1 shows two examples for the mapping between the lifted predicates and the standard predicates. The complete list of mappings for the lifted predicates that are defined in [8] can be found in [11]. Clearly, the list is extensible for the lifted predicates that can be introduced in the future.

Table 1: Mapping lifted predicates into standard predicates.

Lifted Predicate	Standard Predicate
mregion **intersects** *region*	*mregion* **passes** *region*
speed(*mpoint*) > *real*	**sometimes** (**speed**(*mpoint*) > *real*)

In Table 1, the *intersects* lifted predicate yields an *mbool* which is *true* during the time intervals where the *mregion* argument spatially intersects the *region* argument. The corresponding *passes* predicate yields *true* if the *mregion* argument ever intersects with the *region* argument. Clearly, if *passes* yields *false*, so does the STPP as well. The benefit of adding *passes* is that the optimizer will already have rules for its optimization. The second row in Table 1 demonstrates a rather more general mapping. The *sometimes* predicate accepts an *mbool* and yields *bool*. It yields *true* if its argument is ever true, otherwise *false*. Hence, *sometimes* is a general way to map lifted predicates into standard predicates. This is done with the hope that the query optimizer already has rules for optimizing *sometimes*(.). In our implementation, we've extended the optimizer with such rules.

To illustrate the idea, the following query shows how the *snow storm* query is rewritten.

```
SELECT id, storm
FROM SnowStorm
WHERE
  pattern([ storm intersects Berlin as onBerlin,
      storm intersects Dresden as onDresden,
      speed(center(storm)) >= 80.0 as highSpeedB,
      speed(center(storm)) >= 80.0 as highSpeedD] ],
  [stconstraint(onBerlin, onDresden,  vec(aabb)),
   stconstraint(highSpeedB, onBerlin, vec(abba, abb.a)),
   stconstraint(highSpeedD, onDresden, vec(abba, abb.a)])
  and
  storm passes Berlin and
  storm passes Dresden and
  sometimes(speed(center(storm)) >= 80.0) and
  sometimes(speed(center(storm)) >= 80.0
```

where the four additional standard predicates in the end of the query will trigger the optimizer to use available spatial and spatiotemporal indexes on the *storm* attribute during the generation of the query execution plan.

5. GROUP STP

This is the second part of the PhD project. It is still under progress. The goal is to design a language for group STP queries, integrate it with the query optimizer, and do the implementation in Secondo.

We model group STPs as compound patterns that consist of primitive patterns, denoted *patternoids*. For example, the trend-setting pattern is a compound pattern that consist of the two patternoids the *trend-setters* and the *followers*.

Accordingly, the language that we propose describes the pattern in three steps: the *lifted predicates*, the *patternoids*, and the *pattern*. The lifted predicates, similar to their use in the STPP, map the moving objects data to *mbool* values.

The *patternoid* operators report primitive group patterns by processing the *mbool* results form the lifted predicates. The *pattern* is the top level layer which is represented by the *reportpattern* operator. It applies temporal constraints to the results of the patternoid operators and report the found instances of the group STP.

5.1 Patternoid Operators

We have defined two patternoid operators in our design: the *gpattern* and the *crosspattern*. The *gpattern* operator reports the *concurrency* patterns. These are the patterns that are described in terms of sets of moving objects that concurrently maintain similar values for certain movement attributes. For example, 40 moving objects that move concurrently with speed more than 80 km/h for a period of 10 minutes is a concurrency pattern.

The *crosspattern* operator reports the patterns that are described in terms of the interactions between the moving objects. For example the *flock* pattern is described as a group of at least n moving objects, where the mutual spatial distance between every pair is at most r for a time duration of at least d. A slightly different pattern is the *moving cluster*. It requires that every moving object in the group has at least one neighbor in the same group with a distance of at most r. Both patterns are expressed based on the mutual distance between pairs of moving objects.

Obviously the list of patternoid operators is extensible. Formally a patternoid operator is defined as:

DEFINITION 8. A *patternoid operator* is a mapping

$$\underline{set}(\underline{tuple}(\underline{identifier}, \underline{moving}(\underline{x}))) \rightarrow \underline{set}(\underline{mset})$$

□

where the tuple-set/stream represents the moving objects of any moving type and their identifiers. Every *mset* in the result represents a time dependent set of the identifiers of a group of moving objects that fulfill the patternoid. An *mset* in the result must contain exactly one *unit* (*uset*) or many adjacent units without temporal gaps in between as we explain later. An *uset* is a constant temporal unit consisting of a time interval and a *set* value, as in Definition 1. The time interval member is denoted *timeInterval*, and the *set* member is denoted *constValue*.

The patternoid operators are allowed to accept other arguments in order to do this mapping. One important argument for example is the *lifted predicate* that maps the moving objects into *mbool*s for further processing.

In the following subsection, we formally define the *gpattern* patternoid operator. The definition of the *crosspattern* operator is omitted here due to the lake of space.

5.2 The gpattern Operator

The *gpattern* operator has two variants:

- gpattern(M, α, n, d, *alteast*).
- gpattern(M, α, n, d, *exactly*).

where M is the tuple-set in Definition 8, α is a lifted predicate applied to the moving objects in M, $n > 0$ is an *int*, and $d > 0$ is a time *duration* (e.g. in seconds).

In the following definitions, X denotes an *mset*, x denotes a *uset*, P denotes a time *periods*, I denotes a time *interval*, t denotes a time *instant*, and e denotes a moving object.

DEFINITION 9. gpattern(M, n, d, α, *atleast*)

$$
\begin{aligned}
= R = \{(X) | & \text{always}(X \subseteq M),\ \text{always}(|X| \geq n), \\
& \text{nocomponents}(\text{deftime}(X)) = 1, \\
& \nexists Y \in R \ \text{ such that } \ \text{deftime}(X) \subseteq \text{deftime}(Y), \\
& \forall x \in X, \forall e \in x.\text{constValue} \ \text{ such that} \\
& \quad P = \text{deftime}(e \in X) : \\
& \ i) \forall I \in P : \text{length}(I) \geq d \\
& \ ii) \forall t \in P : (\alpha(e))(t) \\
& \}
\end{aligned}
$$

□

where R is the *set*(*mset*) result of the operator. The first two conditions state that every *mset* in result always contains at least n moving objects from the input. The third condition states that a result will contain exactly one *unit* (*uset*) or many adjacent units without gaps in between. To guarantee a finite number of results, the fourth condition assures that the number of members and the definition time of every reported group are the maximum possible. The last condition states that a moving object can appear in the pattern several times (i.e. join and leave the group several times). Every *join* must be for a duration of at least d. The operators *always*, *nocomponents*, *deftime*, ...etc that are used in the definition are defined in [8].

The operator reports all groups of at least n objects, that concurrently fulfill the lifted predicate α for a time period of at least d. The group members may change (i.e. some members join or leave the group), but their number does not go below n. The second variant of the operator is defined as:

DEFINITION 10. gpattern(M, n, d, α, *exactly*)

$$
\begin{aligned}
= \{(X) | & \text{always}(X \subseteq M),\ \text{always}(|X| = n), \\
& \text{nocomponents}(X) = 1,\ \text{length}(\text{deftime}(X)) \geq d, \\
& \forall e \in X[0].\text{constValue}, \forall t \in X[0].\text{timeInterval} : (\alpha(e))(t), \\
& \forall I \ \text{ such that } \ X[0].\text{timeInterval} \subset I : \\
& \quad \exists e \in X[0].\text{constValue}, \exists t \in I \ \text{ such that } \ \neg(\alpha(e))(t) \\
& \}
\end{aligned}
$$

□

where this variant reports all possible groups of size n. The group members are not allowed to change, therefore every *mset* in the result contains exactly one *uset*.

The number of results from this variant can be very large. For example, if a group of size $k > n$ is found that fulfills the pattern, the number of results is $\binom{k}{n}$. We are not determined yet whether to ignore this variant during the implementation, or to keep it as an option for the user.

The *gpattern* operator works conceptually similar to the *REMO* model discussed in Section 2. It can, however, process continuous trajectories. It also overcomes the expensive space requirement of the *REMO* matrix. The well defined inputs and return types of the patternoid operators, allows for seamless integration within the DBMS.

Similar to the *REMO* model, the *gpattern* operator does not take into consideration the spatial proximity between the group members. We have defined operators for the *mset* to check the spatial proximity among the members. These operator can be used to filter the results of the *gpattern*

similar to the REMO approach. A better solution however, is to use the *crosspattern* operator because it inherently support all kinds of mutual relationships between group members including the spatial proximity.

5.3 The reportpattern Operator

The *reportpattern* operator is the top level operator for reporting group STPs. It is similar to the STPP in Section 4.1 in that it allowes for temporal constraints between the results of the patternoid operators. Its signature is:

$$
\begin{aligned}
& set(tuple[identifier, moving(x)]) \\
& \times list\ of\ (fun(.) \rightarrow set(mset)) \\
& \times list\ of\ bool \\
& \rightarrow set(tuple[mset, mset, ...])
\end{aligned}
$$

where the second argument is the list of patternoid operators each of which has an alias. The first argument is passed as is to each of the patternoid operators in the second argument. The third argument is a list of temporal constraints similar to those in Definitions 4, 5, and 6. The *tuple*s in the result contain as many attributes as the number of patternoid operators. The attributes names are the aliases of the patternoid operators, and they have the same order.

Similar to the STPP, the operator is modeled as a binary temporal constraint satisfaction problem. The variables are the aliases of the patternoid operators. The domain of each variable is the definition times of the *mset*s yielded by the corresponding patternoid operator. Note that the definition time of each of these *mset*s is a single time interval according to Definition 8. The constraints of the CSP are the list of temporal connectors in the last argument. Once a *complete supported assignment* (i.e. an assignment of all variables that fulfills all the constraints) is found, the corresponding *mset*s are put into a *tuple* and added to the result stream.

The following example shows the use of the *reportpattern* and the *gpattern* operators to express a *trend-setting* pattern query.

```
query Ghazals feed
reportpattern[
  trendSetter: gpattern(speed(.Trip) > 30.0, 5,
    create_duration(0, 60000), atleast),
  followers:   gpattern(speed(.Trip) > 30.0, 40,
    create_duration(0, 180000), atleast );
  stconstraint("trendSetter", "follower",
    vec("abab", "aba.b", "abba"))]
consume
```

where *Ghazals* is a relation with the schema:
Ghazals[Id: *int*, Trip: *mpoint*]
The query is written in the SECONDO executable language. This is because we didn't study the integration with the query optimizer yet, hence no SQL-like syntax. The first line feeds the *Ghazals* relation as a stream of tuples. The *reportpattern* operator gets the tuple stream, two patternoid operators with the aliases *trendSetter* and *followers*, and one temporal constraint. The *trendSetter* patternoid reports all groups of at least 5 ghazals that run with speed more than 30 km/h for a duration of at least one minute. The *followers* patternoids report similar groups of minimum size of 40 and minimum duration of 3 minutes. The temporal constraint accepts only the combinations such that the *followers* patternoid starts after the *trendSetter* patternoid and disappears after, with, or before it.

6. CONCLUSIONS

This PhD project aims at defining a language for spatiotemporal pattern queries. The work is divided into two parts: individual STP queries, and group STP queries. The two proposed languages are highly expressive compared to other closely related work. They are designed to fit within the context of spatiotemporal DBMSs, and to integrate with the query optimizers.

7. ACKNOWLEDGMENTS

This PhD project is covered by the DAAD scholarship granted to the author in the year 2008.

8. 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] G. Andrienko, N. Andrienko, and S. Wrobel. Visual analytics tools for analysis of movement data. *SIGKDD Explor. Newsl.*, 9(2):38–46, 2007.

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

[5] J. A. Cotelo Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. Algorithms for moving objects databases. *Comput. J.*, 46(6):680–712, 2003.

[6] M. Erwig and M. Schneider. Spatio-temporal predicates. *IEEE Trans. on Knowl. and Data Eng.*, 2002.

[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] 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.*, 2000.

[9] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras. Complex spatio-temporal pattern queries. In *VLDB '05*, 2005.

[10] P. Laube. *Analysing point motion Spatio-temporal data mining of geospatial lifelines.* PhD thesis, Department of Geography, University of Zurich, Switzerland, 2005.

[11] M. Sakr and R. H. Güting. Spatiotemporal pattern queries. Technical Report Informatik-Report 355, FernUniversität Hagen, November 2009.

[12] M. Sakr and R. H. Güting. A new approach for spatiotemporal pattern queries in trajectory databases. In *MDM '10: Proceedings of the 11th International Conference on Mobile Data Management*, May 2010.