Spatiotemporal Pattern Queries in SECONDO

Mahmoud Attia Sakr and Ralf Hartmut Güting

Databases for New Applications, Fernuniversität Hagen, Germany
{mahmoud.sakr,rhg}@fernuni-hagen.de

Abstract. We describe an initial implementation for spatiotemporal pattern queries in SECONDO. That is, one can specify for example temporal order constraints on the fulfillment of predicates on moving objects. It is shown how the query optimizer is extended to support spatiotemporal pattern queries by suitable indexes.

Keywords: Spatiotemporal pattern queries, SECONDO, Query optimizer.

1 Introduction

Moving objects are objects that change their position and/or extent with time. Having the trajectories of these objects stored in a suitable database system allows for issuing spatiotemporal queries. One can query for example for animals which crossed a certain lake during a certain time.

Spatiotemporal pattern queries (STPQ) provide a more complex query framework for moving objects. They are used to query for trajectories which fulfill a certain sequence of predicates during their movement. For example, suppose predicates *P*, *Q* and *R* that can hold over a time interval or a single instant. We would like to be able to express spatiotemporal pattern conditions like the following:

- *P* then (later) *Q* then *R*.
- *P* ending before 8:30 then *Q* for no more than 1 hour.
- (*Q* then *R*) during *P*.

The predicates *P*, *Q*, *R*, etc. might be of the form

- Vehicle *X* is inside the gas station *S*.
- Vehicle *X* passed close to the landmark *L*.
- The speed of air plane *Z* is between 400 and 500 km/h.

STPQs are the intuitive way to query the moving objects by their movement profile. We demonstrate a novel approach for STPQs along with its implementation in SECONDO. Some other approaches can be found in [3], [4], [5] and [6].

2 SECONDO Platform

SECONDO [1] is an extensible DBMS platform that doesn't presume a specific database model. Rather it is open for new database model implementations. It consists of three loosely coupled major components: the kernel, the GUI and the query optimizer.

It supports two syntaxes: an SQL-like syntax and a special syntax called *SECONDO executable language*.

All three SECONDO components can be extended. The kernel, for example, is extended by algebra modules. In an algebra module one can define new data types and/ or new operations. In this work, we added the *spatiotemporal pattern algebra*.

Due to its clean architecture, SECONDO is suitable for experimenting with new data types, database models, query types, optimization techniques and database operations. The source code and documentation for SECONDO are available for download [1].

3 Spatiotemporal Pattern Queries

A main idea underlying our approach to spatiotemporal pattern queries is to reuse the concept of *lifted predicates* [2]. Basically by this concept we define time dependent versions of static predicates. A static binary predicate has the signature

$$\sigma_1 \times \sigma_2 \rightarrow \underline{bool}$$

where σ_1 and σ_2 are of static types (i.e: *point*, *int* or *region*). Lifted counterparts of this predicate are obtained by replacing one or both parameters by their moving versions. The signature of a lifted predicate is one of the following:

$$\underline{moving}(\sigma_1) \times \sigma_2 \rightarrow \underline{moving}(\underline{bool})$$

$$\sigma_1 \times \underline{moving}(\sigma_2) \rightarrow \underline{moving}(\underline{bool})$$

$$\underline{moving}(\sigma_1) \times \underline{moving}(\sigma_2) \rightarrow \underline{moving}(\underline{bool})$$

Since the arguments are time dependent, the result is time dependent as well.

SECONDO uses the sliced representation for moving data types. For a *moving*(*bool*), for example, the sliced representation could be thought of as an array of units with every unit consisting of a time interval and a boolean value. Therefore, each unit is a mapping between a time interval/ slice and a value that holds during this time. Units are not allowed to overlap in time.

For many static predicates lifted counterparts are already implemented in SECONDO. We use lifted predicates in formulating patterns. Hence we can easily leverage a considerable part of the available infrastructure.

A simple STPQ may be

```
Find all cars that entered a gas station, parked close to the
bank, and then drove away fast (candidate bank robbers).
```

This can be expressed in SQL as follows:

```
SELECT * FROM car AS c, landmark As l WHERE l.type= "gas
station" and pattern(c.trip inside l.region then
distance(c.trip, bank)< 0.2 then speed(c.trip)> 100.0)
```

where *c.trip* is a *moving*(*point*) storing the trajectory of the car, the distance is measured in km and the speed in km/h. This pattern consists of three predicates connected to each other by the *then* temporal connector. In our context, *then* is a binary temporal connector with the meaning that the second parameter must start to be true any time greater than or equal to the time when the first parameter starts to be true. In the following illustration we call the three predicates *P*, *Q* and *R* in order. They are lifted predicates that return a moving(bool) since one of the parameters to these predicates is of a moving type.

At the executable level of SECONDO we introduce a new predicate (operator) *stpattern*. Its arguments are a tuple and a sequence of lifted predicates (i.e., functions mapping the tuple into a moving(bool)).

Fig. 1 shows how the *stpattern* operator works applied to the sequence <*P*, *Q*, *R*>. The time is indicated on the horizontal axis and the results of evaluating the three predicates are shown on the vertical axis. The top most result *P*, for example, is a moving(bool) consisting of three units: false before t_1, true between t_1 and t_2 and false again after t_2. Note that we draw only the true intervals.

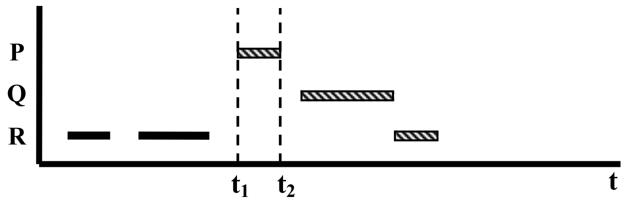


Fig. 1. Evaluating the result of a STPQ for one tuple

The pattern is fulfilled if and only if all the results of the predicates in the sequence have true units and the temporal order of these units meets the order in the sequence. In this example, the order of the sequence is *P* then *Q* then *R* and it is fulfilled by the three hatched intervals in Fig. 1.

The *stpattern* operator is a lazy operator. It evaluates the predicates in order. Once it is sure that the sequence of lifted predicates is not fulfilled (e.g., one predicate is always false) it returns *false*. This makes the operator more efficient by skipping the evaluation of unnecessary predicates. For example, if *P* is always false then *Q* and *R* are not evaluated.

More complex pattern descriptors may include temporal connectors like *later*, *meanwhile* or *immediately* or temporal constraints in the form of *P for 10 min then Q for no more than an hour meanwhile R*. We have ongoing research which considers such ideas but these are not yet included in this first implementation.

4 Query Optimization for Spatiotemporal Pattern Queries

SECONDO is a complete system in which even the query optimizer is accessible and, in fact, relatively easily extensible. Hence in contrast to previous work we are able to actually integrate pattern queries into the overall optimization framework. Obviously for an efficient execution of pattern queries on large databases the use of indexes is mandatory. We now consider how pattern predicates can be mapped to efficient index accesses.

The basic idea is to add each of the lifted predicates in a modified form as an extra "standard predicate" to the query, that is, a predicate returning a boolean value. The optimizer then should have for the rewritten predicate already some optimization rule available ("translation rules" in SECONDO) to map it into efficient index access.

For example, consider the lifted predicate "*c.trip inside l.region*" with moving(point) *c.trip*. We can rewrite it to a standard predicate "*c.trip passes l.region*" returning a bool value. The optimizer then already has rules to find trajectories passing the given region if their units are appropriately indexed.

A general way to convert a lifted predicate into a standard predicate is to apply the operation *sometimes*. It is a SECONDO predicate that takes a moving(bool) and returns a bool. It returns *true* if its parameter ever assumes *true* during its lifetime, otherwise *false*. So the strategy we implemented is to rewrite each lifted predicate *P* into a predicate *sometimes*(*P*) and to add translation rules that were missing.

Here is a brief description of how the optimizer processes STPQs along with the required extensions:

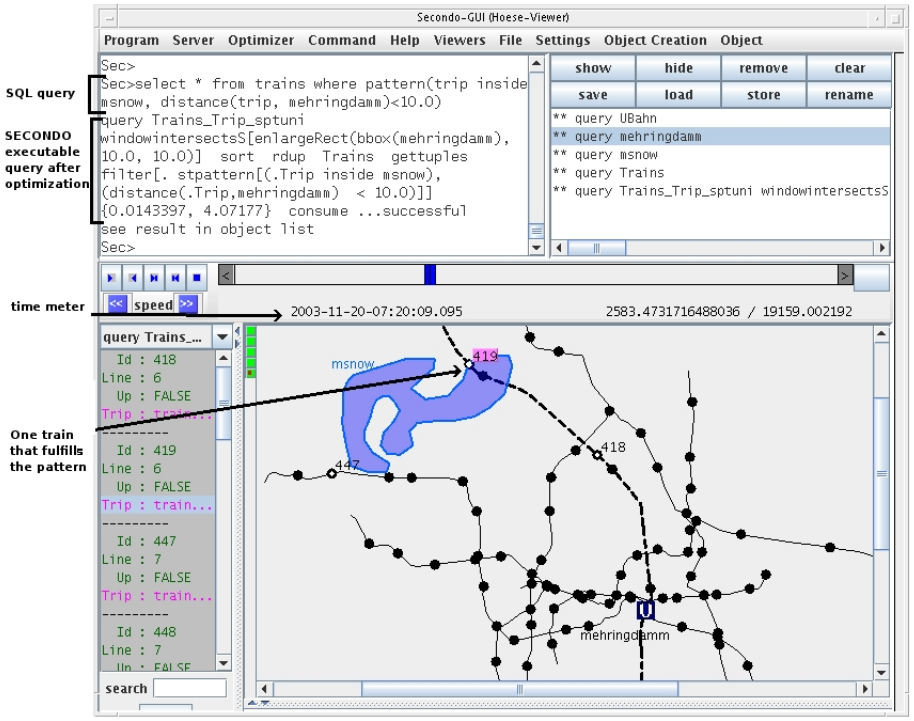
1. The parser parses the SQL-like query. For that we extended the parser to accept the syntax of the pattern operator.
2. The *query rewriting* rules are invoked. We added rules that for every predicate *P* in the pattern descriptor add a condition *sometimes*(*P*) to the where-clause of the query. Clearly the rewritten query is equivalent to the original one. We keep a list of these additional conditions for further processing.
3. We added translation rules for *sometimes*(*Q*) terms that were not yet available.
4. The optimizer continues its cost based optimization procedure trying to utilize available indexes.
5. After the best plan is chosen, the optimizer invokes the rules for translating the best plan into the *SECONDO executable language* and passes the query to the kernel for execution. A rule is added here to remove the additional *sometimes* predicates (with the help of the maintained list described in step 2) before passing the query to the kernel.

5 What Will Be Demonstrated

In this demo we will run several spatiotemporal pattern queries in SECONDO. The database used in the demonstration is the *berlintest* database coming with the SECONDO distribution containing geodata and objects of types moving(point) and moving(region). We will also demonstrate the query optimizer and the process of optimizing the STPQs. SECONDO will be invoked in its complete configuration where the three modules (kernel, GUI, and optimizer) are running together.

In Fig. 2, the GUI displays the result of an STPQ. The query is first written in SQL-like syntax. The GUI passes the SQL query to the optimizer. The optimizer generates a query plan which is passed back to the GUI (note that the plan accesses an index called Trains_Trip_sptuni). The GUI passes the executable query to the kernel, the kernel executes the query, the kernel passes the results to the GUI. Finally, the GUI tries to find the best viewer available and renders the results (in this case the Hoese- Viewer is chosen).

In Fig. 2, the trains fulfilling the STPQ are displayed as white circles with black outlines. The rest of the trains are black circles. "trip" is a moving(point) and "msnow" is a moving(region). They change their locations with time (see the time meter). The train with id 419 fulfills the pattern. It moves downwards and its path is indicated by a dashed line.


Secondo-GUI (Hoese-Viewer)
Program Server Optimizer Command Help Viewers File Settings Object Creation Object
show hide remove clear
save load store rename
SQL query
SECONDO executable query after optimization
time meter
One train that fulfills the pattern
speed
search
msnow
419
418
447
mehringdamm

```
Sec>
Sec>select * from trains where pattern(trip inside
msnow, distance(trip, mehringdamm)<10.0)
query Trains_Trip_sptuni
windowintersectsS[enlargeRect(bbox(mehringdamm),
10.0, 10.0)]  sort  rdup  Trains  gettuples
filter[. stpattern[(.Trip inside msnow),
(distance(.Trip,mehringdamm)  < 10.0)]]
{0.0143397, 4.07177}  consume ...successful
see result in object list
Sec>
```

```
** query UBahn
** query mehringdamm
** query msnow
** query Trains
** query Trains_Trip_sptuni windowintersectsS
```

```
2003-11-20-07:20:09.095          2583.4731716488036 / 19159.002192
```

```
query Trains_...
  Id : 418
Line : 6
  Up : FALSE
Trip : train...
---------
  Id : 419
Line : 6
  Up : FALSE
Trip : train...
---------
  Id : 447
Line : 7
  Up : FALSE
Trip : train...
---------
  Id : 448
Line : 7
  Up : FALSE
```

Fig. 2. A sample spatiotemporal pattern query

References

1. SECONDO home page, `http://dna.fernuni-hagen.de/Secondo.html/index.html`
2. Güting, R.H., Böhlen, M.H., Erwig, M., Jensen, S.J., Lorentzos, N.A., Schneider, M., Vazirgiannis, M.: A foundation for representing and quering moving objects. ACM Transactions Database Systems 25(1), 1–42 (2000)
3. Du Mouza, C., Rigaux, P.: Mobility Patterns. In: 2nd Workshop on Spatio-Temporal Database Management, STDBM 2004 (2004)
4. Erwig, M.: Toward Spatiotemporal Patterns. In: De Caluwe, et al. (eds.) Spatio-Temporal Databases, pp. 29–54. Springer, Heidelberg (2004)
5. Schneider, M.: Evaluation of Spatio-Temporal Predicates on Moving Objects. In: 21st Int. Conf. on Data Engineering, ICDE, pp. 516–517 (2005)
6. Hadjieleftheriou, M., Kollios, G., Bakalov, P., Tsotras, V.J.: Complex Spatio-Temporal Pattern Queries. In: VLDB 2005, pp. 877–888 (2005)