Spatiotemporal Pattern Queries

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

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

June 30, 2010

Abstract

This paper presents a novel approach to express and evaluate the complex class of queries in moving object databases called *spatiotemporal pattern queries* (STP queries). That is, one can specify temporal order constraints on the fulfillment of several predicates. This is in contrast to a standard spatiotemporal query that is composed of a single predicate. We propose a language design for spatiotemporal pattern queries in the context of spatiotemporal DBMSs. The design builds on the well established concept of *lifted predicates*. Hence, unlike previous approaches, patterns are neither restricted to specific sets of predicates, nor to specific moving object types. The proposed language can express arbitrarily complex patterns that involve various types of spatiotemporal operations such as range, metric, topological, set operations, aggregations, distance, direction, and boolean operations. This work covers the language integration in SQL, the evaluation of the queries, and the integration with the query optimizer. We also propose a simple language for defining the temporal constraints. The approach allows for queries that were never available. We provide a complete implementation in C++ and Prolog in the context of the SECONDO platform. The implementation is made publicly available online as a SECONDO Plugin, which also includes automatic scripts for repeating the experiments in this paper.

1 Introduction

The area of moving objects databases has been active since the early 2000s, and is recently receiving a lot of interest because of the advances in the positioning and sensor technologies that generates large amounts of moving objects data. These databases deal with the geometries that change over time, also called spatiotemporal data. There are two classes of models for such data. The first deals with the current movement and the predicted near future (e.g. [25]). These models are optimized for cheaper updates. The second class deals with the trajectories or the history of the movement (e.g. [19]), and these models are optimized for cheaper queries. In this paper, we focus on the second class of models, the trajectory databases.

Having the spatiotemporal trajectories of the moving objects stored in a database system allows for issuing spatiotemporal queries. One can query, for example, for animals which crossed a certain lake during a certain time interval or for the total length of a car trajectory inside a certain zone. There has been a lot of work on providing spatiotemporal data management and query operations (e.g. [8]). Recently more focus is given to the nearest neighbor queries (e.g. [18], [14]), and the trajectory similarity queries (e.g. [23]).

However, due to the recent application domains, trajectories are getting longer. Additionally, due to the privacy restrictions, trajectories are getting anonymized. The precise position and/or extent of the moving objects are more and more replaced by the events or the changes that happened during the movement, the so called *semantic trajectories* [6]. It is difficult to query, for example, sequences of such changes of data using traditional spatiotemporal queries. This difficulty comes from the fact, that they are composed of one predicate. In many cases, one would need to express temporal orders (relative or absolute) of several changes, each of which need to be expressed as a predicate. For example, *find all trains that encountered a delay of more than half an hour after passing through a snow storm* is a query that expresses two changes/predicates, one happening after the other. It is very difficult if not impossible to express such a query using the traditional spatiotemporal query methods.

Spatiotemporal pattern (STP) queries provide a more complex query framework for moving objects. In particular, they specify temporal order constraints among a set of time-dependent predicates. For example, suppose the predicates P, Q, and R that may hold over one or more time intervals and/or instants. We would like to be able to express 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, and R, etc. might be of the form:

- Vehicle X on road W.
- Train X is inside a snow storm Y.
- The extent of the storm area Y is larger than 4 square kms.
- The speed of air plane Z is between 400 and 500 km/h.

For such conditions to hold, there must exist a time interval for each of the predicates, during which it is fulfilled, and this set of time intervals must fulfill the temporal order in the condition. The spatiotemporal patterns described by such conditions cannot be expressed by traditional spatiotemporal queries. One would rather need the *spatiotemporal pattern queries*.

More about the importance of STP queries in many fields of application is illustrated in [10]. So far we are talking about the spatiotemporal patterns that occur within individual trajectories. That is every trajectory in the database can individually answer the pattern without knowledge of other trajectories. The term Spatiotemporal Patterns is also used in the literature to refer to *group patterns*. This is more related to the spatiotemporal data mining literature. The methods analyze simultaneous movements and the interaction between objects (e.g. patterns like leadership, play, fighting, migration, trend-setting, ... etc). The research in this direction aims at developing a toolbox of data mining algorithms and visual analytic techniques for movement analysis. For example, algorithms for the flock, leadership, convergence and encounter patterns are presented in [15]. In this paper, we are focusing on the individual spatiotemporal pattern queries, simply denoted spatiotemporal pattern queries (STP queries) during the rest of the paper.

Few proposals exist for handling STP queries as will be detailed in the related work section. All of them lack generality in the patterns that can be expressed. They are limited to certain moving objects types (*moving points* in most of the proposals), and to certain types of spatiotemporal predicates (*spatial* predicates and *nearest neighbor* predicates). The approach described in this paper, expresses and evaluates STP patterns that are neither restricted to certain types of moving objects, nor to certain types of predicates. Our contributions are the following:

- The proposed approach is based on a very general and powerful class of predicates, the so-called lifted predicates [19]. They are very powerful as they are simply the time dependent version of arbitrary static predicates. Instead of returning a <u>*bool*</u> value (like standard predicates) they return a $\underline{moving}(\underline{bool})$ (time dependent booleans as defined later). Our approach allows one to formulate temporal constraints on the results of arbitrary expressions returning such moving booleans. Formulating STP queries over lifted predicates allows for a wide range of queries that are not addressed before.

- The proposed approach can be easily extended to support more complex patterns. Section 6 describes one such extension.
- In contrast to previous work we are able to actually integrate STP queries into the query optimizer. Obviously for an efficient execution of pattern queries on large databases the use of indexes is mandatory. In Section 7 we consider how STP queries can be mapped by the query optimizer to efficient index accesses.
- We propose a simple language for describing the relationship between two time intervals (e.g. Allen's operators). The language makes it easier, from the user point of view, to express interval relations without the need to memorize their names.
- The complete implementation of the work in this paper is done in the context of the SECONDO platform [4]. It is publicly available as a SECONDO Plugin and can be downloaded from the Plugins web site [1]. Parallel to this paper, we have written a user manual describing how to install and run the Plugin within a SECONDO system.
- There are automatic scripts for repeating the experiments in this paper. They are installed during the installation of the Plugin. Section 11 describes the procedure to repeat the experiments. The scripts, together with the well documented source code provided in the Plugin, allow the readers to explore our approach, further elaborate on it, and compare with other approaches.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 gives a brief background about the moving objects databases and recalls some necessary definitions from previous work. In Section 4, we define the proposed language. Section 5 formalizes the spatiotemporal pattern predicate as a constraint satisfaction problem, and explains the evaluation algorithms. In Section 6, the basic spatiotemporal pattern predicate is extended into a more expressive version. In Section 7 we show how to integrate our approach seamlessly with the query optimizers. Section 8 is dedicated to the technical aspects of the implementation in the SECONDO framework. The experimental evaluation is shown in Section 9. In Section 10, we demonstrate two application examples that emphasize the expressive power of our approach. Section 11 and the Appendices at the end of the paper describe the experimental repeatability. Finally we conclude in Section 12.

2 Related Work

A theory and a design for spatiotemporal pattern queries, although important, are not yet well established. Only few proposals exist. In [22], a model that relies on a discrete representation of the spatiotemporal space is presented. The 2D space is partitioned in a finite set of user defined partitions, called zones, each of which is assigned a label. The time domain is partitioned into constant-sized intervals. The trajectories are represented as strings of labels. For example, the trajectory part *rzzzh* represents a moving object that stayed in zone *r* for one time unit, moved to zone *z* and stayed there for three time units, then moved to zone *h* for one time unit. The user query is composed as a formal expression, which is then evaluated using efficient string matching techniques.

This approach is not general in the sense that the space and time have to be partitioned. The partitioning depends on the intended application and has to be done in advance. Moreover, only patterns that describe the changes of the location of moving points can be expressed. The approach leaves behind all other kinds of predicates (e.g. topological, metric comparisons, ...) as well as other types of moving objects (e.g. moving regions).

In [20], an index structure and efficient algorithms to evaluate STP queries that consist of spatial and neighborhood predicates is presented. The work addresses the problem of conjoint neighborhood queries (e.g. find all objects that were as close as possible to A at time T_1 then were as close as possible to B at time T_2). The two NN conditions in this query have to be evaluated conjointly. In other words, an object which minimizes the sum of the two distances at the two time points is the answer of this query.

Again the approach addresses only limited types of predicates, and handles moving points only. It tightly couples the evaluation of the predicates with the evaluation of the STP query itself. On the one hand, this allows for efficient evaluation of the STP query. It also allows for the conjoint neighborhood queries, which are not possible in our appraoch for example. On the other hand, it is very specific to this set of predicates. In order to support other predicates and/or other data types, one has to find a way to extend their evaluation algorithms. In the context of systems, a modular design that decouples the predicate evaluation from the STP query evaluation would be preferred.

The series of publications [11], [12], [10], and [24] provide a concrete formalism for spatiotemporal developments. A spatiotemporal development is a composite structure built as an alternating sequence of spatiotemporal and spatial predicates, and they are themselves spatiotemporal predicates. They describe the change, wrt. time, in the spatial relationship between two moving objects. Consider, for example, a moving point MP and a moving region MR. The development $MP\ Crosses\ MR$ is defined as:

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

where *meet* is a spatial predicate that yields true when its two arguments touch each other, and *Disjoint* is a spatiotemporal predicate that yields true when its two arguments are always spatially disjoint. The spatiotemporal predicates, denoted by being capitalized, differ from the spatial predicates in that, the former hold at time intervals while the later hold at instants. Spatiotemporal developments consider two spatiotemporal objects and precisely describe the change in their topological relationship.

The spatiotemporal developments in their definition are not equivalent to spatiotemporal patterns, as they can only describe the change in the topological relationship between two objects. This is not general enough to describe STPs. A natural way of describing STPs would involve several interactions between one trajectory and many other objects in the spatiotemporal space, as well as the trajectory's own motion attributes (e.g. speed, direction, ...etc.).

Additionally, all the related works discussed above share two limitations. First, they do not address issues of system integration and query optimization (e.g. SQL style syntax). Second, only sequential patterns are allowed. A pattern is not allowed to include, for example, concurrent predicates. As shown in the rest of this paper, our approach overcomes these limitations. Mainly, it is designed with expressiveness, system integration, and extensibility in mind.

3 Moving Objects Databases

In previous work [19], [13], and [8], a model for representing and querying moving objects is proposed. The work is based on abstract data types (ADT). The *moving* type constructor is used to construct the moving counterpart of every static data type. Moving geometries are represented using three abstractions; $\underline{moving}(\underline{point})$, $\underline{moving}(\underline{region})$ and $\underline{moving}(\underline{line})$. Simple data types (e.g. $\underline{integer}$, $\underline{bool}$, $\underline{real}$) are also mapped to *moving* types. In the *abstract model* [19], moving objects are modeled as temporal functions that map time to geometry or value. For example, moving points are modeled as curves in the 3D space (i.e. time to the 2D space).

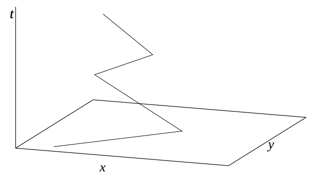
In [13] a discrete data model implementing the abstract model is defined. For all data types in the abstract model, corresponding *discrete* types whose domains are defined in terms of finite representations are introduced. In the discrete model, moving types are represented by the sliced representation as units.

Definition 1 A data type $\underline{moving}(\underline{\alpha})$ is a set of units. Every unit is a pair $(I, \underline{Instant} \rightarrow \alpha)$. 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 $\underline{Instant} \rightarrow \alpha$. 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. $\underline{mpoint}$ denotes $\underline{moving}(\underline{point})$). Similarly, the unit types are denoted by appending *u*. The $\underline{mpoint}$, for example, is modeled in the discrete model as a set of $\underline{upoint}$s, each of which consists of a time interval and a line function. This is illustrated in Figure 1. The coordinates of the $\underline{mpoint}$ at any time instant within the interval are obtained by evaluating the line function. The *moving* type constructor is similarly applied to the scalar data types (e.g. $\underline{real}$, $\underline{string}$, $\underline{bool}$) [19]. A precise definition of the $\underline{mbool}$ data type is given in Section 4.

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

[”2003-11-20-06:06” ”2003-11-20-06:06:08.692”[, (16229.0 1252.0), (16673.0 1387.0)
[”2003-11-20-06:06:08.692” ”2003-11-20-06:06:24.776”[, (16673.0 1387.0), (16266.0 1672.0)
[”2003-11-20-06:06:24.776” ”2003-11-20-06:06:32.264”[, (16266.0 1672.0), (16444.0 1818.0)
[”2003-11-20-06:06:32.264” ”2003-11-20-06:06:39.139”], (16444.0 1818.0), (16144.0 2227.0)



The model offers a large number of operations that fall into three classes:

1. Static operations over the non-moving types. Examples are the topological predicates, set operations and aggregations.

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

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

Lifted operations are obtained by a mechanism called *temporal lifting* [19]. All the static operations defined for non-moving types are uniformly and consistently made applicable to the corresponding moving types. 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

$$T_1 \times .... \times T_n \rightarrow \underline{bool}$$

where T_i is a type variable that can be instantiated by any static/non-temporal data type (e.g. $\underline{integer}$, $\underline{point}$, $\underline{region}$). □

Example: BrandenburgGate *inside* Berlin.

Definition 3 A *lifted predicate* is a function with the signature

$$T_1 \times .... \times T_k \times \uparrow T_{k+1} \times ... \times \uparrow T_n \rightarrow \underline{mbool}$$

where $\uparrow$ is the *moving* type constructor. A lifted predicate is, hence, obtained by allowing one or more of the parameters of a static predicate to be of a *moving* data type. Consequently, the return type is a time dependent boolean $\underline{mbool}$. □

Example: Train_RE1206 *inside* Berlin.
Note that *inside* in this example is a lifted predicate because the *Train_RE1206* is a moving object. It is therefore different from the standard *inside* predicate in the previous example.

4 Spatiotemporal Pattern Predicates

The *Spatiotemporal Pattern Predicate* (STP predicate) is the tool that we propose for expressing STP queries. It describes the pattern as a set of *time-dependent predicates* that are fulfilled in a certain temporal arrangement (e.g. a sequence). To motivate the idea of our design, consider the following example:

Example: A query for possible bank robbers may look for the cars which entered a gas station, kept close to the bank for a while, then drove away fast.

The query describes an STP consisting of three time-dependent predicates: *car inside gas station*, *car close to the bank*, and *speed of car $\geq$ 80 km/h*. The predicates are required to be fulfilled in a sequential temporal order.

We propose a modular language design of the STP predicate. It consists of two parts. The first defines a special kind of predicates that accept moving object arguments and report the time intervals, during which they are fulfilled. The second part is to define a language for temporal constraints on the predicate fulfillments.

Fortunately, the *lifted predicates* [19] in Definition 3 do exactly what is needed in the first part. Lifted predicates yield objects of type $\underline{mbool}$, which tell about the time intervals of the predicate fulfillment. Moreover, they are not restricted to certain data types of arguments nor to certain types of operations. Formulating the STP predicate on the top of the lifted predicates easily leverages a considerable part of the available infrastructure. The temporal constraints, in the second part, enforce certain temporal arrangements between the $\underline{mbool}$ results of the lifted predicates.

We start here by a rough illustration. The details follow later in this section. The bank robbers query is written as follows:

```
SELECT c.licencenumber
FROM cars c, landmark l
WHERE l.type = "gas station" and
  pattern([ c.trip inside l.region as gas,
    distance(c.trip, bank) < 50.0 as bnk,
    speed(c.trip) > 80000 as leaving],
  [stconstraint(gas, bnk, vec(aabb)),
   stconstraint(bnk, leaving, vec(abab, aa.bb, aabb)])
```

where *c.trip* is an $\underline{mpoint}$ that stores the car's trajectory. The STP predicate, denoted *pattern* in the SQL-like syntax, includes a set of three lifted predicates:

```
c.trip inside l.region,
distance(c.trip, bank) < 50.0,
speed(c.trip) > 80000
```

having the aliases *gas*, *bnk*, and *leaving*. The syntax of the STP predicate assigns *aliases* for the lifted predicates, so that they can be referred to in the temporal constraints. This is analogous to the aliases given to attributes and tables in the standard SQL. An alias of a lifted predicate can be any valid unique identifier. The STP predicate in this example includes two temporal constraints, denoted *stconstraint* in the SQL-like syntax. Each constraint is stating a temporal relationship between two of the lifted predicates (i.e. binary temporal constraints). The syntax *vec*(.) states the temporal order between the fulfillments of the two lifted predicates. Roughly speaking, the first temporal constraint states that the car came close to the bank after it has left the gas station. The second constraint is a bit more tricky. We wish to say that the car left the bank area quickly. This means that the car started fast, or may have started normally and then sped up after a while. The three arguments to the *vec*(.) operator state these three possibilities, as formalized later in this section.

Now we start the formal definition of the STP predicate. We first recall the definition of the $\underline{mbool}$ data type from [13]. Let *Instant* denote the domain of time instants, isomorphic to $\mathbb{R}$. Let I_T be the set of possible time intervals, i.e:

$$\begin{aligned} I_T = \{(t_1, t_2, lc, rc) | t_1, t_2 \in \mathit{Instant}, \\ lc, rc \in \{\text{false, true}\}, t_1 < t_2, \\ (t_1 = t_2) \Rightarrow lc = rc = \text{true}\} \end{aligned}$$

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, see [13]. Let the domain of *Boolean Unit* <u>*ubool*</u> be:

$$\mathit{UBool} = \{(i, u) | i \in I_T, u \in \{\mathit{false}, \mathit{true}\}\}$$

and the domain of <u>*mbool*</u> is:

$$\begin{aligned} \mathit{MBool} = \{U \subset \mathit{UBool} \mid \forall (i_1, u_1), (i_2, u_2) \in U : \\ (i) \;\; i_1 = i_2 \Rightarrow u_1 = u_2 \\ (ii) \;\; i_1 \neq i_2 \Rightarrow i_1 \cap i_2 = \emptyset \wedge \\ i_1 \text{ adjacent } i_2 \Rightarrow u_1 \neq u_2\} \end{aligned}$$

where i_1 adjacent i_2 $:\Leftrightarrow i_1.t_2 = i_2.t_1 \wedge (i_1.rc \vee i_2.lc)$. This last condition ensures the <u>*mbool*</u> objects have a unique representation, the one with the minimum number of units.

Following we define a language for temporal relationships between pairs of time intervals. It will be the base for the temporal constraints between the lifted predicates inside the STP predicate. In the temporal logic literature some studies define the relationships between pairs of time intervals, and assign them names (e.g. the 13 Allen's operators [5]). Here we propose a language, instead of names. This is because, in our case 26 such relationships are possible, which makes it difficult for a user to memorize the names. Table 1 shows the 26 terms of this language, and a graphical illustration of each. In the terms, the letters aa denote the begin and end time instants of the first interval. Similarly bb are the begin and end of the second interval. The order of letters describes the temporal relationship, that is, a sequence ab means $a < b$. The dot symbol denotes the equality constraint, hence, the sequence $a.b$ means $a = b$, and $a.a$ means that the start and the ends of the first interval are the same (i.e. the interval degenerates into a time instant).

Formally, let IR be the set of interval relationships of Table 1, that is

$$\mathit{IR} = \{\text{aabb, abba, ..., a.a.b.b}\}$$

Let i_1, $i_2 \in I_T$, $ir = s_1 s_2 ... s_k \in \mathit{IR}$ (note that $4 <= k <= 7$, that is, the shortest term includes two *a*'s and two *b*'s, and the longest term includes additionally three dots),

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

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

Table 1: A language for expressing interval relationships

Term	Illustration	Term	Illustration	Term	Illustration
Both arguments are intervals (Allen's operators)					

aabb	`aaaa`
	`      bbbb`
		abba	`aaaaaaaa`
			`  bbbb`
				bbaa	`      aaaa`
					`bbbb`

a.bab	`aaaa`
	`bbbbbbbb`
		aa.bb	`aaaa`
			`    bbbb`
				a.bba	`aaaaaaaa`
					`bbbb`

bb.aa	`    aaaa`
	`bbbb`
		baa.b	`    aaaa`
			`bbbbbbbb`
				abab	`aaaa`
					`  bbbb`

aba.b	`aaaaaa`
	`  bbbb`
		baba	`  aaaa`
			`bbbb`
				a.ba.b	`aaaa`
					`bbbb`

baab	`  aaaa`
	`bbbbbbbb`				

The first argument is an instant					

a.abb	`a`
	`  bbbb`
		bb.a.a	`   a`
			`bbbb`
				a.a.bb	`a`
					`bbbb`

bba.a	`      a`
	`bbbb`
		ba.ab	`  a`
			`bbbb`		

The second argument is an instant					

b.baa	`  aaaa`
	`b`
		aa.b.b	`aaaa`
			`    b`
				b.b.aa	`aaaa`
					`b`

aab.b	`aaaa`
	`     b`
		ab.ba	`aaaa`
			`  b`		

Both arguments are instants					

a.ab.b	`a`
	`  b`
		b.ba.a	`  a`
			`b`
				a.a.b.b	`a`
					`b`

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

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

The *vec*(.) in the SQL-like syntax allows for composing such SI subsets. For syntactic elegance, one can assign names to them, and use the names in the queries. This is done using the *let* statement as follows:

```
let then = vec(abab, aa.bb, aabb);
let later = vec(aabb);

SELECT c.licencenumber
FROM cars c, landmark l
WHERE l.type = "gas station" and
  pattern([c.trip inside l.region as gas,
    distance(c.trip, bank) < 50.0 as bnk,
    speed(c.trip) > 80000 as leaving],
  [stconstraint(gas, bnk, later),
   stconstraint(bnk, leaving, then])
```

That is, *later* and *then* can hence be used inside the *stconstraint* operator.

For ease of presentation, in the following we define the STP predicate within the relational data model. The definitions can however be adapted easily to fit within other database models (e.g. object oriented), thanks to the ADT modeling of the moving objects which does not depend on a particular database model.

Let *tuple* denote a tuple type in the sense of the relational data model[1]. Let D_{tuple} denote the domain

[1]Here *tuple* is viewed as a type variable that can be instantiated by any valid tuple type.

of the tuples conforming to this type. Let the domain of the type $\underline{mbool}$ be:

$$D_{\underline{mbool}} = MBool$$

A *time-dependent predicate* is a function with signature:

$$tuple \rightarrow \underline{mbool}$$

hence it is a function

$$f : D_{tuple} \rightarrow D_{\underline{mbool}}$$

We denote a predicate with this signature as p_{tuple}, and a set of such predicates as P_{tuple} when the tuple type is relevant.

Note that the definition of a *time-dependent predicate* is more general than that of a *lifted predicate*. A lifted predicate also yields an $\underline{mbool}$, but it must correspond to some standard static predicate, see Definition 3. Formally, the STP predicate is composed of a set of time-dependent predicates, and a set of temporal constraints, as shown later in this section. Throughout the text, however, we are often using the term *lifted predicate* instead of the more general term *time-dependent predicate* because the former seems more relevant from the user point of view. That is, users will be using lifted predicates to compose their STP queries. This will become obvious from the many query examples in the rest of this paper.

Let $P_{tuple} = \{p_1, ..., p_n\}$ be a set of time-dependent predicates. A temporal constraint on P_{tuple} is an element of the set:

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

Hence it is a binary temporal constraint, that assigns a pair of predicates in P_{tuple} a set of interval relationships. In the SQL-like syntax, the operator *stconstraint* expresses a temporal constraint. It accepts three arguments: two *aliases* of time-dependent predicates, and a set of interval relationships composed by the *vec*(.) operator.

Based on the above definitions, a spatiotemporal pattern predicate is defined as follows:

Definition 4 A *spatiotemporal pattern predicate* (*STP predicate*) is a pair (P_{tuple}, C), where $C \subseteq TC(P_{tuple})$. □

In SQL, the operator *pattern* denotes the spatiotemporal pattern predicate. For an STP predicate to hold, all the temporal constraints in C must be fulfilled. Formally it is as follows:

Let $t \in D_{tuple}$ be a tuple and $P_{tuple} = \{p_1, ..., p_n\}$, we denote by $p_k(t)$ the evaluation of $p_k \in P_{tuple}$ on t. Hence $p_k(t) \in MBool$. We also define the set of *candidate assignments* $CA(P_{tuple}, t)$ as:

$$CA(P_{tuple}, t) = p_1^{true} \times ... \times p_n^{true}$$

where $p_k^{true} = \{i | (i, true) \in p_k(t)\}$. That is, the $CA(P_{tuple}, t)$ is simply the Cartesian product of the sets of time intervals during which the time-dependent predicates in P_{tuple} are fulfilled with respect to the tuple t.

Let $ca = (i_1, ..., i_n) \in CA(P_{tuple}, t)$ and let $c = (j, k, SI) \in TC(P_{tuple})$ be a temporal constraint

$$ca \text{ fulfills } c :\Leftrightarrow i_j \text{ and } i_k \text{ fulfill } SI$$

Let $C \subseteq TC(P_{tuple})$ be a set of temporal constraints, and let $t \in D_{tuple}$ be a tuple. The set of *supported assignments* of C is defined as:

$$SA(P_{tuple}, C, t) = \{ca \in CA(P_{tuple}, t) \mid \forall c \in C : ca \text{ fulfills } c\}$$

That is, for a *candidate assignment* to be a *supported assignment*, it must fulfill all the constraints in C. An STP predicate is fulfilled for a given tuple if and only if such a supported assignment is found.

Definition 5 A *spatiotemporal pattern predicate* is a function with the signature $tuple \rightarrow \underline{bool}$. Given a tuple t of type *tuple*, its evaluation is defined as:

$$eval((P_{tuple}, C), t) = (SA(P_{tuple}, C, t) \neq \emptyset)$$

□

5 Evaluating Spatiotemporal Pattern Predicates

The formalization of the STP predicate in the previous section maps pretty well into the well known Constraint Satisfaction Problem (CSP). This section illustrates this mapping and the algorithms used to evaluate the STP predicate.

Definition 6 Formally, a *constraint satisfaction problem* is defined as a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a set of initial domains, and C is a set of constraints. Each variable $x_i \in X$ has a non-empty domain $d_i \in D$. CSP algorithms remove values from the domains during evaluation once it is discovered that the values cannot be part of a solution. Each constraint involves a subset of variables and specifies the allowable combinations of values for this subset. An assignment for a subset of variables is *supported* if it satisfies all constraints. A solution to the CSP is in turn a *supported assignment* of all variables. □

Recalling, from Definition 4, that the STP predicate contains the set $P_{tuple} = \{p_1, ..., p_n\}$ of time-dependent predicates, a straight forward way to construct the sets X, D of the CSP is as follows:

1. For every $p_i \in P_{tuple}$, define a variable x_i with the same name as the *alias* of p_i in the user query. Set $X := X \cup x_i$.
2. Given a tuple t of type *tuple*, compute for every $p_i \in P_{tuple}$ its evaluation $p_i(t)$.
3. For every $p_{ij}^{true} \in p_i^{true}$, set $D_i := D_i \cup p_{ij}^{true}$.

That is, a CSP variable is created for every time-dependent predicate in the STP predicate. The aliases of the lifted predicates, as specified in the user query, are used as the variable names. The *initial domain* of every CSP variable is the set of time intervals during which the corresponding time-dependent predicate is fulfilled. Finally the set of constraints in the CSP is the same as the set of constraints in the STP predicate. As is shown next, this is not exactly how we map the STP predicate into a CSP. The main difference is that the domains of the variables (i.e. the set D) are evaluated in a lazy fashion. Following, we briefly discuss the known algorithms for solving CSPs. Later in this section, we will be proposing another algorithm for evaluating the CSP that fits more with our approach.

A CSP having only binary constraints is called *binary CSP* and can be represented graphically in a *constraints graph*. The nodes of the graph are the variables and the links are the binary constraints. Two nodes are linked if they share a constraint. The neighborhood of a variable in the constraints graph are all variables that are directly linked to it. The spatiotemporal pattern predicate is fulfilled if and only if its corresponding CSP has at least one supported assignment.

CSPs are usually solved using variants of the backtracking algorithm. The algorithm is a depth-first tree search that starts with an empty list of assigned variables and recursively tries to find a solution (i.e. a supported assignments of all variables). In every call, backtracking adds a new variable to its list and tries all the possible assignments. If an assignment is supported, a new recursive call is made. Otherwise the algorithm backtracks to the last assigned variable. The algorithm runs in exponential time and space.

Constraint propagation methods [7] (also called local consistency methods) can reduce the domains before backtracking to improve the performance. Examples are the ARC Consistency and Neighborhood Inverse Consistency (NIC) algorithms. They detect and remove some values from the variable domains that cannot be part of a solution. Local consistency algorithms do not guarantee backtrack-free search. To have the nice property of backtrack-free search one would need to enforce *n*-consistency (equivalent to global consistency), which is again exponential in time and space.

The solvers for CSPs assume that the domains of the variables are known in advance. This is, however, a precondition that we wish to avoid. In the STP predicate, calculating the domain of a variable is equivalent to evaluating the corresponding lifted predicate. Since this can be expensive, we wish to delay the evaluation of the domains.

The proposed algorithm *Solve Pattern* below tries to solve the sub-CSP of $k-1$ variables (CSP_{k-1}) first and then to extend it to CSP_k. Therefore, an early stop is possible if a solution to the CSP_{k-1} cannot be found. Which means that, in case no solution is found, the evaluation will be stopped as soon as this is realized, without the uncessary evaluation of the remaining lifted predicates.

The *Solve Pattern* algorithm uses three data structures: the SA list (for Supported Assignments), the *Agenda* and the *Constraint Graph*. The Agenda keeps a list of variables that are not yet consumed by the algorithm. One variable from the Agenda is consumed in every iteration. Every supported assignment in the SA list is a solution for the sub-CSP consisting of the variables that have been evaluated so far. In iteration k there are $k-1$ previously evaluated variables and one newly evaluated variable (X_k with domain D_k). Every entry in SA at this iteration is a solution for the CSP_{k-1}. To extend the SA, the Cartesian product of SA and D_k is calculated. Then only the entries that constitute a solution for CSP_k are kept in SA. CSP_k is constructed using the consumed variables and their corresponding constraints in the constraint graph.

Algorithm Solve Pattern
```
input:  variables, constraints
output:  whether the $CSP$ consistent or not

  1. Clear $SA$, Agenda and Constraint Graph
  2. Add all variables to Agenda
  3. Add all constraints to the Constraint Graph
  4. WHILE Agenda not empty
     (a) Pick a variable $X_i$ from the Agenda
     (b) Calculate the variable domain $D_i$ (i.e.  evaluate the
         corresponding lifted predicate)
     (c) Extend $SA$ with $D_i$
     (d) IF $SA$ is empty return NotConsistent
  5. return Consistent
```

Algorithm Extend
```
input:  i, $D_i$; the index and the domain of the newly evaluated variable

  1. IF $SA$ is empty
     (a) FOREACH interval $I$ in $D_i$
          i. INSERT a new row sa in $SA$ having sa[i]= $I$ and
             undefined for all other variables
     ELSE
     (a) set $SA$ = the Cartesian product $SA \times D_i$
     (b) Construct the subgraph $CSP_k$ that involves the
         variables in $SA$ from the Constraint Graph.
     (c) FOREACH sa in $SA$
          i. IF sa does not satisfy the $CSP_k$, remove sa from $SA$
```

The methodology for picking the variables from the Agenda has a big effect on the run time. The best method will choose the variables, so that inconsistencies are detected soon. For example, suppose an STP predicate having four predicates with aliases u, v, w, and x. The constraints are:

stconstraint(u, x, vec(abab)), *stconstraint*(v, x, later) , and *stconstraint*(w, x, vec(bb.a.a)).

If the variables are picked in sequential order u, v, w, then x, the space and time costs are the maximum. Since u, v, and w are not connected by any constraints, the SA is populated by the Cartesian product of their domains in the first three iterations. The actual filter to SA starts in the fourth iteration after x is picked.

The function that picks the variables from the Agenda chooses the variables according to their *connectivity rank* in the Constraint Graph. The connectivity rank of a variable is the summation of its individual connectivities in the Constraint Graph. If a given variable is connected to an Agenda variable with a constraint, it gets 0.5 connectivity score for this constraint. This means that evaluating this variable contributes 50% in evaluating the constraint because the other variable is still not evaluated. If the other variable in the constraint is a non-Agenda variable (i.e. a variable that is already evaluated), the connectivity score is 1. Back again to the example, in the first iteration, the variables u, v, and w have connectivity ranks of 0.5, whereas x has 1.5. Therefore, x is picked in the first iteration. In the second iteration u, v, and w have equal connectivity ranks of 1, so the algorithm picks any of them.

This variable picking methodology tries to maximize the number of evaluated constraints in every iteration with the hope that they filter the SA list and detect inconsistencies as soon as possible.

The time cost of the *Solve Pattern* algorithm is

$$\sum_{i=1}^{n} \prod_{k=1}^{i} d_k \times e_k$$

where n is the number of variables, d_k is the number of values in the domain of the k^{th} variable and e_k is the number of constraints in CSP_k. The storage cost is

$$\sum_{i=1}^{n} \prod_{k=1}^{i} d_k$$

The algorithm runs in $O(ed^n)$ and takes $O(d^n)$ space.

The exponential time and space costs are not prohibitive in this case. This is because the calculations done within the iterations are simple comparisons of time instants. Moreover, the number of variables in an STP query is expected to be less than 8 in the normal case. The *Solve Pattern* algorithm is more focused on minimizing the number of evaluated lifted predicates (statement 4.b of the algorithm). The cost of evaluating the lifted predicates varies, but it is expected to be expensive because the evaluation usually requires retrieving and processing the complete trajectory of the moving object. The run time analysis of many lifted predicates is illustrated in [8].

6 Extending the Definition of the STP Predicates

Back to the example of bank robbers, a sharp eyed reader will notice that the provided SQL statement can retrieve undesired tuples. Suppose that long enough trajectories are kept in the database. A car that entered a gas station in one day, passed close to the bank in the next day, and in a third day sped up will be part of the result. To avoid this, we would like to constrain the period between leaving the gas station till speeding up to be at most 1 hour.

Indeed the proposed design is flexible so that such an extension is easy to integrate. The idea is that after the STP predicate is evaluated, the SA data structure contains all the supported assignments. As illustrated before, a supported assignment assigns an interval to each lifted predicate during which it is satisfied. At the same time the interval values of all variables satisfy all the constraints in the STP predicate. Now that we know the time intervals, we can impose more constraints on them. For example, we state that the period between leaving the gas station (first predicate) till speeding up (third predicate) must be at most 1 hour.

The following describes formally an extended version of the STP predicate that allows for such additional constraints. Let $P_{tuple} = \{p_1, ..., p_n\}$ be a set of time-dependent predicates, and let $C \subseteq TC(P_{tuple})$ be a set of temporal constraints. Let g be a function:

$$g : I_T^n \times D_{tuple} \rightarrow D_{\underline{bool}}$$

That is, g is a predicate that accepts a set of n time intervals and a *tuple*, and yields a *bool*.

An extended STP predicate is defined as follows:

Definition 7 An *extended spatiotemporal pattern predicate* is a triple (P_{tuple}, C, g). Given a tuple t of type *tuple* its evaluation is defined as:

$$eval((P_{tuple}, C, g), t) = (\{sa \in SA(P_{tuple}, C, t)| \;\; g(sa, t) = true\} \neq \emptyset)$$

□

That is, the boolean predicate g is applied to the supported assignments in SA and to the input tuple t. For the *extended STP predicate* to be fulfilled, g must be fulfilled at least once. The evaluation of the extended STP predicate is, hence, done in two parts, that both must succeed. The first solves the STP predicate (P_{tuple}, C) for the given tuple t, and the second part, which is processed only after the success of the first part, evaluates the boolean predicate g for every supported assignment. Hence, conditions on the list of supported assignments SA are possible.

Syntactically, the user is provided with two functions *start*(.) and *end*(.) that yield the start and end time instants of the intervals in an SA element. The two functions are in the form:

$$f : I_T^n \times \{1, ..., n\} \rightarrow Instant$$

Given a supported assignment $sa \in SA$ and an *index*, the two functions yield the start and the end time instants of the time interval at this index in sa.

Formally let $sa = \{i_1, ..., i_n\} \in SA(P_{tuple}, C, t)$.

$$start(sa, k) = i_k.t_1, \text{ and}$$
$$end(sa, k) = i_k.t_2$$

where $1 \leq k \leq n$.

To implement the extension, step 5 in the *Solve Pattern* algorithm is changed to `return` SA. The predicate g is then iteratively evaluated for the elements of the SA. The algorithm of evaluating the extended STP predicate is not shown in the paper, because it is a trivial change for the *Solve Pattern* algorithm.

The extended STP predicate is denoted *expattern* in the SQL-like syntax. The bank robbers query is rewritten using it as follows:

```
SELECT c.licencenumber
FROM cars c, landmark l
WHERE l.type = "gas station" and
  patternex([c.trip inside l.region as gas,
    distance(c.trip, bank) < 50.0 as bnk,
    speed(c.trip) > 100000 as leaving],
  [stconstraint(gas, bnk, later),
   stconstraint(bnk, leaving, then],
  start(leaving) - end(gas) < 1)
```

where the additional condition `start(leaving) - end(gas) < 1` ensures that the time period between the car getting out from the gas station (i.e. *end*(*gas*)) till it starts leaving the bank area (i.e. *start*(*leaving*)) is less than one hour. Note that in the SQL-like syntax, the *start*, and *end* operators get the predicate *aliases*, rather their indexes as in the definition.

More complex conditions can be expressed. The time intervals can be used, for example, to retrieve parts from the moving object trajectory to express additional spatial conditions. For example, the query for possible bank robbers may more specifically look for the cars which entered a gas station, made a round or more surrounding the bank, then drove away fast. To check that the car made a round surrounding the bank, a possible solution is to check the part of the car trajectory close to the bank for self intersection. The query may be written as follows

```
SELECT c.licencenumber
FROM cars c, landmark l
WHERE l.type = "gas station" and
  patternex([c.trip inside l.region as gas,
    distance(c.trip, bank) < 50.0 as bnk,
    speed(c.trip) > 100000 as leaving],
  [stconstraint(gas, bnk, later),
   stconstraint(bnk, leaving, then],
  isSelfIntersecting(
    trajectoryPart(c.trip, start(bnk), end(bnk))) and
  (start(leaving) - end(bnk)) < 1)
```

where *trajectoryPart* computes the spatial trajectory of the moving object between two time instants and *isSelfIntersecting* checks a line for self intersection.

7 Optimizing Spatiotemporal Pattern Predicates

In Section 5 we explained the evaluation of the spatiotemporal pattern predicate. The proposed algorithm is efficient because it avoids the unnecessary evaluation of lifted predicates. In the context of large-scale DBMS, this is not enough. Obviously for an efficient execution of pattern queries on large databases the use of indexes is mandatory. It should be triggered by the query optimizer during the creation of the executable plans.

In this section, we demonstrate a generic procedure for integrating the STP predicate with query optimizers. We do not assume a specific optimizer or optimization technique. The optimizer is however required to have some basic features that will probably be available in any query optimizer. In the following subsection, we describe these basic assumptions.

7.1 Query Optimization

A typical query optimizer contains two basic modules; the *rewriter* and the *planner* [21]. The rewriter uses some heuristics to transform a query into another equivalent query that is, hopefully, more efficient or easier to handle in further optimization phases. The planner creates for the user query (or the rewritten version) the set of possible *execution plans* (possibly restricted to some classes of plans). Finally it applies a selection methodology (e.g. cost based) to select the best plan.

We assume that the query optimizer contains the rewriter and the planner modules. We also assume that it supports the data types and operations on moving objects, in SQL predicates as described in [19] and [13].

7.2 Query Optimization for Spatiotemporal Pattern Predicates

One observation that we like to make clear is that the STP predicate itself does not process database objects directly. Instead, the first operation applied is the evaluation of the lifted predicates that compose the STP predicate. The idea, hence, is to design a general framework for optimizing the lifted predicates within the STP predicate. This framework should trigger the optimizer to use the available indexes for the currently supported lifted predicates as well as for those that might be added in the future. It should utilize the common index structures. Although specialized indexes, as in [20], can achieve higher performance, the overhead of maintaining them within a system is high and they only serve specific purposes, which makes them unfavorable in the context of systems.

The 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 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 done during query rewriting. The additional standard predicates in the rewritten query trigger the planner to use the available indexes. To illustrate the idea, the following query shows how the bank robbers query in Section 4 is rewritten.

```
SELECT c.licencenumber
FROM cars c, landmark l
WHERE l.type = "gas station" and
  pattern([c.trip inside l.region as gas,
    distance(c.trip, bank) < 50.0 as bnk,
    speed(c.trip) > 100000 as leaving],
  [stconstraint(gas, bnk, later),
   stconstraint(bnk, leaving, then])
  and
  c.trip passes l.region and
  sometimes(distance(c.trip, bank) < 50.0) and
  sometimes(speed(c.trip) > 100000)
```

The three lifted predicates in the STP predicate `x inside y`, `distance(x, y) < z`, and `speed(x) < y` are mapped to the standard predicates `x passes y`, `sometimes(distance(x, y) < z)`, and `sometimes(speed(x) < y)`, respectively. Here *sometimes*(.) is a predicate that accepts an <u>*mbool*</u> and yields true if the argument ever assumes true during its lifetime, otherwise false. Each of the standard predicates ensures that the corresponding lifted predicate is fulfilled at least once, a necessary but not sufficient condition for the *pattern* predicate to be fulfilled. Clearly, the rewritten query is equivalent to the original query.

The choice of the standard predicate depends on the type of the lifted predicate and the types of the arguments. For example, the lifted spatial range predicates (i.e. the spatial projection can be described by a box) are mapped into the *passes* standard predicate. The passes predicate [19], in this example, is fulfilled if the car `c.trip` ever passed the gas station `l.region`. If *passes* fails, then we know that *inside* is never true and that *pattern* will also fail. The planner should have for the added passes predicate already some optimization rule available (e.g. use a spatial R-tree index when available). In Section 9.2.2 we show an optimized query written in the SECONDO executable language.

To generalize this solution, we define a table of mappings between the lifted predicates (or groups of them) and the standard predicates. Clearly, this mapping is extensible for the lifted predicates that can be introduced in the future. The mapping for the set of lifted predicates proposed in [19] is shown in Table 2.

For the lifted spatial range predicates, they map into *passes* and the available translation rules for passes do the rest. The $distance(x, y) < z$ is conceptually equivalent to a lifted spatial range predicate, where the spatial range is the minimum bounding box of the static argument extended by z in every side. Other types of lifted predicates are mapped into *sometimes*. We need to provide translation rules that translate *sometimes*(.) into index lookups. For every type of lifted predicates, one such translation rule is required. For example, the *sometimes*(*Pred*), where *Pred* is a lifted left range predicate, searches for a B-tree defined on the units of the moving object, and performs a left range search in the B-tree. We show examples for these translation rules within SECONDO in Section 8.2.

Table 2: Mapping lifted predicates into standard predicates.

Lifted Predicates	**Type**	**Standard Predicates**
σ = α	lifted spatial range	σ **passes** α
$\underline{mpoint} \times \underline{point} \rightarrow \underline{mbool}$		
$\underline{mregion} \times \underline{region} \rightarrow \underline{mbool}$		
σ **inside** α		
$\underline{mpoint} \times \underline{region} \rightarrow \underline{mbool}$		
$\underline{mpoint} \times \underline{points} \rightarrow \underline{mbool}$		
$\underline{mpoint} \times \underline{line} \rightarrow \underline{mbool}$		
$\underline{mregion} \times \underline{region} \rightarrow \underline{mbool}$		
$\underline{mregion} \times \underline{points} \rightarrow \underline{mbool}$		
$\underline{mregion} \times \underline{line} \rightarrow \underline{mbool}$		
σ **intersects** α		
$\underline{mregion} \times \underline{points} \rightarrow \underline{mbool}$		
$\underline{mregion} \times \underline{region} \rightarrow \underline{mbool}$		
$\underline{mregion} \times \underline{line} \rightarrow \underline{mbool}$		
σ = α	lifted equality	**sometimes(**σ = α**)**
$\underline{mint} \times \underline{int} \rightarrow \underline{mbool}$		
$\underline{mbool} \times \underline{bool} \rightarrow \underline{mbool}$		
$\underline{mstring} \times \underline{string} \rightarrow \underline{mbool}$		
$\underline{mreal} \times \underline{real} \rightarrow \underline{mbool}$		
$\sigma <= \alpha, \sigma < \alpha$	lifted left range	**sometimes(**$\sigma <= \alpha$**)**, **sometimes(**$\sigma < \alpha$**)**
$\underline{mint} \times \underline{int} \rightarrow \underline{mbool}$		
$\underline{mbool} \times \underline{bool} \rightarrow \underline{mbool}$		
$\underline{mstring} \times \underline{string} \rightarrow \underline{mbool}$		
$\underline{mreal} \times \underline{real} \rightarrow \underline{mbool}$		
$\sigma >= \alpha, \sigma > \alpha$	lifted right range	**sometimes(**$\sigma >= \alpha$**)**, **sometimes(**$\sigma > \alpha$**)**
$\underline{mint} \times \underline{int} \rightarrow \underline{mbool}$		
$\underline{mbool} \times \underline{bool} \rightarrow \underline{mbool}$		
$\underline{mstring} \times \underline{string} \rightarrow \underline{mbool}$		
$\underline{mreal} \times \underline{real} \rightarrow \underline{mbool}$		
distance(σ **,** α**)** $<$ threshold	lifted spatial range	σ **passes** enlargeRect(bbox(α), threshold, threshold)
$\underline{mpoint} \times \underline{region} \rightarrow \underline{mreal}$		
$\underline{mpoint} \times \underline{point} \rightarrow \underline{mreal}$		
$\underline{mregion} \times \underline{point} \rightarrow \underline{mreal}$		
$\underline{mregion} \times \underline{region} \rightarrow \underline{mreal}$		
Other lifted predicates, P		**sometimes(**P**)**

This two steps optimization helps develop a general framework for optimizing the *sometimes*(.) predicate, which may also appear directly in the user queries. Note that we can alternatively rewrite all lifted predicates into *sometimes*(.), and provide translation rules accordingly. It remains an implementation decision, which approach to use.

8 The Implementation in SECONDO

SECONDO [4], [16], [17] is an extensible DBMS platform that does not presume a specific database model. Rather it is open for new database model implementations. For example, it should be possible to implement relational, object-oriented, spatial, temporal, or XML models.

SECONDO consists of three loosely coupled modules: the kernel, GUI and query optimizer. The kernel includes the command manager, query processor, algebra manager and storage manager. The kernel may be extended by algebra modules. In an algebra module one can define new data types and/or new operations. The integration of the new types and/or operations in the query language is then achieved by adding syntax rules to the command manager.

The SECONDO kernel accepts queries in a special syntax called SECONDO *executable language*. The SQL-like syntax is provided by the optimizer. For more information about SECONDO modules see [4] and [3]. For more information about extending SECONDO see the documentation on [2].

If it is the case that a new data type needs a special graphical user interface (GUI) for display, the SECONDO GUI module is also extensible by adding viewer modules. Several viewers exist that can display different data types. Moving objects, for example, are animated in the *Hoese* viewer with a time slider to navigate forwards and backwards.

A large part of the moving objects database model presented in [19], [13], [8], that we also assume in the paper, is realized in SECONDO. That is, the current SECONDO version 2.9.1 includes the algebra modules, the viewer modules, and the optimizer support for moving objects. In the following subsections, we describe the implementation of the STP predicate in SECONDO 2.9.1. This implementation is available as a SECONDO Plugin as explained in Section 11.

8.1 Extending the Kernel

We have implemented the STP predicate in the SECONDO kernel in a new algebra module called *STPatternAlgebra*. The algebra contains:

1. One data type *stvector*. The class represents a set of interval relationships as defined in Section 4. The SECONDO operator *vec* is used to create an *stvector* instance. The operator accepts a set of strings from Table 1, and constructs the *stvector* instance accordingly.

   Example: `vec("aabb", "a.abb", "a.a.bb")`.

2. The *stconstraint* operator. The operator represents a temporal constraint within the STP predicate. The signature of the operator is:

   $$\underline{string} \times \underline{string} \times \underline{stvector} \rightarrow \underline{bool}$$

   The first and second parameters are the aliases for two lifted predicates.

   Example: `stconstraint("predicate1", "predicate2", vec("a.a.bb"))`.

3. The *stpattern* operator. The operator implements the STP predicate. It has the signature:

   $$\textit{tuple} \times AliasedPredicateList \times ConstraintList \rightarrow \underline{bool}$$

   where the $AliasedPredicateList$ is a list of time-dependent predicates, each of which has an alias, and the $ConstraintList$ is a list of temporal constraints (i.e. a list of $stconstraint$ operators).

4. The *stpatternex* operator. The operator implements the extended STP predicate, Section 6. It has the signature:

   $$\textit{tuple} \times AliasedPredicateList \times ConstraintList \times \underline{bool} \rightarrow \underline{bool}$$

5. The *start*(.) and the *end*(.) operators, described in Section 6. They accept a $\underline{string}$ representing a predicate alias and return the start/end of the corresponding time interval. The operators have the signature:

   $$\textit{string} \rightarrow \underline{instant}$$

Using these operators, the query for bank robbers can be written in SECONDO executable language as follows:

```
query cars feed {c}
landmark feed {l}
  filter[.type_l = "gas station"]
product
filter[.
  stpatternex[gas: .trip_c inside .region_l,
    bnk: distance(.trip_c, bank) < 50.0,
    leaving: speed(.trip_c) > 100000;
  stconstraint("gas", "bnk", vec("aabb")),
    stconstraint("bnk", "leaving", vec("abab", "aa.bb", "aabb"));
  duration2real(start("leaving") - end("gas")) < (1/24) ]]
consume
```

where *feed* is a postfix operator that scans a relation sequentially and converts it into a stream of tuples. The query performs a cross product between the tuples of the *cars* relation and the tuples of *landmark* relation that has the value *"gas station"* in their *type* attribute. The resulting tuple stream after the cross product is filtered using the extended STP predicate *stpatternex*. Finally, the *consume* operator converts the resulting tuple stream into a relation, so that it can be displayed.

8.2 Extending the Optimizer

The SECONDO optimizer is written in Prolog. It implements an SQL-like query language which is translated into an optimized query in SECONDO executable language. The SECONDO optimizer includes a separate rewriting module that can be switched on and off by setting the optimizer options. The planner implements a novel cost based optimization algorithm which is based on *shortest path search in a predicate order graph*. The predicate order graph (POG) is a weighted graph whose nodes represent sets of evaluated predicates and whose edges represent predicates, containing all possible orders of predicates. For each predicate edge from node *x* to node *y*, so-called plan edges are added that represent possible evaluation methods for this predicate. Every complete path via plan edges in the POG from the bottom-most node (i.e. zero evaluated predicates) till the top-most node (i.e. all predicates evaluated) represents a different execution plan. Different paths/execution plans represent different orderings of the predicates and different evaluation methods. The plan edges of the graph are weighted by their estimated costs, which in turn are based on given selectivities. Selectivities of predicates are either retrieved from prerecorded values, or estimated by sending selection or join queries on small samples of the involved relations to the SECONDO kernel and reading the cardinality of the results. The algorithm is described in more detail in [17] as well as in the SECONDO programmers guide [2].

Our extension to the optimizer has three major parts: query rewriting, operator description, and translation rules. In the query rewriting, we choose to rewrite all the lifted predicates into *sometimes*(.). This is because an accurate rewriting based on the mapping in Table 2 requires that we know the data types of the arguments. The SECONDO optimizer knows the data types only after query rewriting is done.

Following are the Prolog rules that do the rewriting:

```
inferPatternPredicates([], []).

inferPatternPredicates([Pred|Preds],
    [sometimes(Pred)|Preds2] ):-
  assert(removefilter(sometimes(Pred))),
  inferPatternPredicates(Preds,Preds2).
```

where the *inferPatternPredicate* accepts the list of the lifted predicates within the STP predicate as a first argument, and yields the a list of rewritten predicates in the second argument. The additional *sometimes*(.) predicates are kept in the table `removefilter(.)`, so that it is possible to exclude them from the executable plan afterwards.

In the operator descriptions, we annotated the lifted predicates by their types (e.g. lifted left range) as in Table 2. Then we provided translation rules for *sometimes*(.) for every type of lifted predicates. Following is an example for such a rule:

```
indexselectLifted(arg(N), Pred ) =>
    gettuples(rdup(sort(windowintersectsS(
    dbobject(IndexName), BBox))),  rel(Name, *))
  :-
  Pred =..[Op, Arg1, Arg2],
  ((Arg1 = attr(_, _, _), Attr= Arg1) ;
   (Arg2 = attr(_, _, _), Attr= Arg2)),
  argument(N, rel(Name, *)),
  getTypeTree(Arg1, _, [_, _, T1]),
  getTypeTree(Arg2, _, [_, _, T2]),
  isLiftedSpatialRangePred(Op, [T1, T2]),
  (
   ( memberchk(T1, [rect, rect2, region, point, line, points, sline]),
     BBox= bbox(Arg1)
   );
   ( memberchk(T2, [rect, rect2, region, point, line, points, sline]),
     BBox= bbox(Arg2)
   )
  ),
  hasIndex(rel(Name, _), Attr, DCindex, spatial(rtree, unit)),
  dcName2externalName(DCindex, IndexName).
```

where this rule translates the *lifted spatial range* predicates into an R-tree window query, as indicated in the rule header. The => operator can be read as *translates into*. It means that the expression to the right is the translation of the expression to the left, if the conditions in the rule body hold. The body of the rule starts by inferring the types of the arguments of the lifted predicate within the *sometimes*(.). Then it uses them to make sure that the predicate is of the type *lifted spatial range*. Finally, it checks whether a spatial R-tree index on the involved relation and attribute is available in the catalog. It tries to find a spatial R-tree built on the units of the moving object. Similar translation rules are provided for other types of indexes. The optimized query in Section 9.2.2 shows the effect of these translation rules.

9 Experimental Evaluation

We proceed with an experimental evaluation of the proposed technique. The intention is to give an insight into the performance. It is clear that the runtime of an STP predicate depends on the number and types of the lifted predicates. Therefore, we show three experiments. The first measures only the overhead of evaluating the spatiotemporal pattern predicate. That is, we set the time of evaluating the lifted predicates to negligible values.

In the second experiment, we generate random STP predicates with varying numbers of lifted predicates and constraints and measure the run time of the queries. The experiment also evaluates the optimization of STP predicates. Every query is run twice; once without invoking the optimizer, and another time with the optimizer being invoked.

The third experiment is dedicated to evaluate the scalability of the proposed approach. It mainly evaluates the proposed optimization approach in large databases. A random set of queries is generated and evaluated againest relations of cardinalities 50,000, 100,000, 200,0000, and 300,000, where the trajectories are indexed using the traditional RTree index.

The first two experiments use the *berlintest* database that is available with the free distribution of SECONDO. The last experiment uses the *BerlinMOD* benchmark [9] to generate the four relations. The benchmark is available for download on [4]. The three experiments are run on a SECONDO platform installed on a Linux machine. The machine is a Pentium-4 dual-core 3.0 GHz processor with 2 GBytes main memory.

9.1 The Overhead of Evaluating STP predicates

To perform the first experiment, we add two operators to SECONDO; *randommbool* and *passmbool*. The operator *randommbool* accepts an *instant* and creates an *mbool* object whose definition time starts at the given time instant, and consists of a random number of units. The operator *passmbool* mimics a lifted predicate. It accepts the name of an *mbool* database object, loads the object and returns it. More details are given below.

9.1.1 Preparing the Data

This section describes how the test data for the first experiment is created. The *randommbool* operator is used to create a set of 30 random *mbool* instances and store them as database objects. The operator creates *mbool* objects with a random number of units varying between 0 and 20. The first unit starts at the time instant provided in the argument. Every unit has a random duration between 2 and 50000 milliseconds. The value of the first unit is randomly set to *true* or *false*. The value of every other unit is the negation of its preceding unit. Hence, the minimal representation requirement [13] of the moving types in SECONDO is met. That is, adjacent units can not be further merged because they have different values.

The 30 *mbool* objects are created by calling `randommbool(now())` 30 consecutive times. This increases the probability that the definition times of the objects temporally overlap.

9.1.2 Generating the Queries

The queries of the first experiment are selection queries consisting of one filter condition in the form of an STP predicate. The queries are generated with different experimental settings, that is, different numbers of lifted predicates and constraints in the STP predicate. The number of lifted predicates varies between 2 and 8. The number of constraints varies between 1 and 16. The queries are not generated for every combination. For example, it does not make sense to generate STP predicates with 2 lifted predicates and 10 constraints. For N lifted predicates, the number of constraints varies between $N-1$ and $2N$. The rationale of this is that, if the number of constraints is less than $N-1$, then the constraint network can not be complete (i.e. some predicates are not referenced within constraints). On the other hand, having more than $2N$ constraints increases the probability of contradicting constraints. For every experimental setting, 100 random queries are evaluated and the average run time is recorded.

A query with 3 lifted predicates and 2 constraints, for example, looks like:

```
query thousand feed
  filter[.
    stpattern[a: passmbool(mb5),
      b: passmbool(mb13),
      c: passmbool(mb3);
    stconstraint("b", "a", later),
      stconstraint("b", "c", vec("abab") ]]
count
```

where `query thousand feed` streams the *thousand* relation, which contains 1000 tuples. For every tuple, the STP predicate *stpattern* is evaluated. Note that the predicate does not depend on the tuples. That is, the same predicate is executed 1000 times in the query. This is to minimize the effect of the time taken by SECONDO to prepare for query execution. The lifted predicates are all in the form of `passmbool(X)`, where `X` is one of the 30 stored random *mbool* objects.

The constraints are generated so that the constraint graph is complete. We start by initializing a set called *connected* having one randomly selected alias. For every constraint, the two aliases are randomly chosen from the set of aliases in the query, so that at least one of them belongs to the set *connected*. The other alias is added to the set *connected* if it was not already a member. After the required number of constraints is generated, we check the completeness of the graph. If it is not complete, the process is repeated till we get a connected graph. The temporal connector for every constraint is randomly chosen from a set containing 31 temporal connectors namely, the 26 simple temporal connectors in Table 1 and 5 vector temporal connectors (later, follows, immediately, meanwhile, and then) (shown in Appendix A).

Before running the queries, we query for the 30 $\underline{mbool}$ objects so that they are loaded into the database buffer. The measured run times should, hence, show the overhead of evaluating the STP predicates in SECONDO because other costs are made negligible.

9.1.3 Results

The results are shown in Figure 2. The number of lifted predicates is denoted as N. Increasing the number of lifted predicates and constraints in the STP predicate does not have a great effect on the run time. This is a direct result of the early pruning strategy in the *Solve Pattern* algorithm. The results show that the evaluation of STP predicate is efficient in terms of run time.

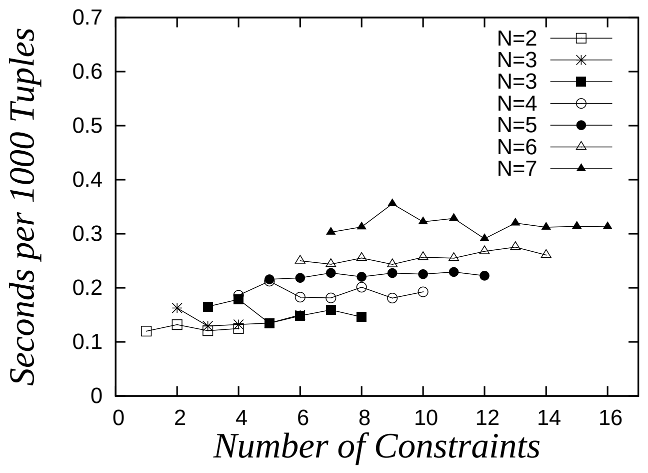


Figure 2: The overhead of evaluating STP predicates

9.2 STP Queries with Optimization

The second experiment is intended to evaluate the run time of STP queries. It also evaluates the effect of the proposed optimization. Unlike the first experiment, the STP predicates in this experiment contain lifted predicates. We generate 10 random queries for every experimental setting and record the average run time. Every query is run twice; without being optimized, and after optimization.

9.2.1 Preparing the Data

The queries use the *Trains20* relation. It is generated by replicating the tuples of the *Trains* relation in the *berlintest* database 20 times. The *Trains* relation was created by simulating the underground trains of the city Berlin. The simulation is based on the real train schedules and the real underground network of Berlin. The simulated period is about 4 hours in one day. The schema of *Trains20* is similar to *Trains* with the additional attribute *Serial*:

Trains20[Serial: $\underline{int}$, Id: $\underline{int}$, Line: $\underline{int}$, Up: $\underline{bool}$, Trip: $\underline{mpoint}$]

where Trip is an $\underline{mpoint}$ representing the trajectory of the train. The relation contains 11240 tuples and has a disk size of 158 MB. To evaluate the optimizer, a spatial R-tree index called *Trains20_Trip_sptuni* is built on the units of the Trip attribute. A set of 300 points is also created to be used in the queries. The points represent geometries of the top 300 tuples in the *Restaurants* relation in the *berlintest* database.

9.2.2 Generating the Queries

The queries are generated in the same way as in the first experiment. In this experiment, however, we use actual lifted predicates instead of *passmbool*. Every lifted predicate in the STP predicate is randomly chosen from

1. distance(trip, *randomPoint*) $<$ *randomDistance*.
2. speed(trip) $>$ *randomSpeed*.

where *randomPoint* is a $\underline{point}$ object selected randomly from the 300 restaurant points, *randomDistance* ranges between 0 and 50, and *randomSpeed* ranges between 0 and 30. The *distance*(., .) $<$. is a sample for the lifted predicates that can be mapped into index access, so that we can evaluate the optimizer. While the queries in the first experiment are created directly in the SECONDO executable language, they are created here in SECONDO SQL. It is an SQL-like syntax that looks similar to the standard SQL, but obeys Prolog rules. The main differences are that everything is written in lower case, and lists are placed within square brackets.

Here is one query example from the generated queries:

```
SELECT count(*)
FROM trains20
WHERE pattern([ distance(trip, point170) < 18.0 as a,
                speed(trip) > 11.0 as b],
              [stconstraint("a", "b", vec("b.ba.a"))])
```

where *pattern* is the SQL operator equivalent to *stpattern* in the executable language. The rewritten version of the query as generated by the rewriting module of the SECONDO optimizer is:

```
SELECT count(*)
FROM trains20
WHERE [ pattern([ distance(trip, point170) < 18.0 as a,
                  speed(trip) > 11.0 as b],
                [stconstraint("a", "b", vec("b.ba.a"))]),
        sometimes(distance(trip, point170) < 18.0),
        sometimes(speed(trip) > 11.0)]
```

Finally, the optimal execution plan is:

```
Trains20_Trip_sptuni
windowintersectsS[ enlargeRect(bbox(point170), 18.0, 18.0)]
sort rdup Trains20  gettuples
  filter[sometimes((distance(.Trip,point170) < 18.0))]
  {0.00480288, 1.69712}
  project[Trip]
  filter[. stpattern[  a: (distance(.Trip, point170) < 18.0),
                       b: (speed(.Trip) > 11.0);
                    stconstraint("a", "b", vec("b.ba.a"))]]
  {0.00480288, 1.49038}
  filter[sometimes((speed(.Trip) > 11.0))]
  {0.883731, 1.48077}
count
```

where the predicates are placed within the *filter*[] operator, which means that they belong to the *where* clause in SQL. The rewriter generates for the two lifted predicates in the original query two standard *sometimes* predicates. The predicate *sometimes*(*distance*(., .) $<$.) is handled by the optimizer as a special kind of range predicate. Since the optimizer can find the spatial R-tree index that we created, it is used. The index access part in the query is:

```
    Trains20_Trip_sptuni windowintersectsS[enlargeRect(., ., .)]
```

This part expands the minimum bounding box of *point170* by the distance threshold value 18.0. The enlarged box is intersected with the R-tree to get the candidate tuple id's. The rest of the query retrieves the data of the candidate tuples and performs the query. The pairs of numbers between the curly brackets do not affect the semantics of the query. They are estimated predicate selectivities and run time statistics used to help estimate the query execution progress.

9.2.3 Results

In Figure 3, the chart to the left shows the average run times of the non-optimized STP queries. The chart to the right shows the average run times of their optimized counterparts. The *N* is again the number of lifted predicates. The run times of the optimized STP predicates are very promising.

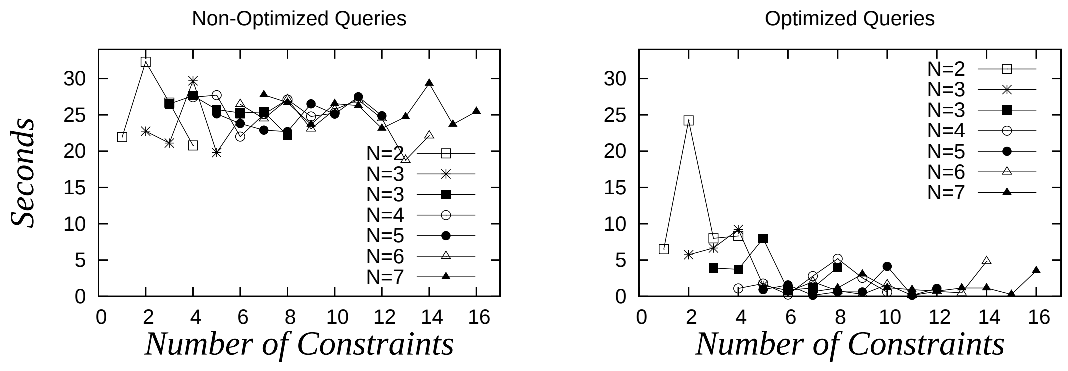


Figure 3: The run times for STP queries on the *Trains20* relation

The high peak in the optimized queries chart at *N = 2* and *Number of Constraints = 2* is because it happened that five of the ten generated queries have only *speed*(.) $<$. predicates. Since the *sometimes*(*speed*(.) $<$.) predicate does not map into index access, the average run time for this experimental setting is close to the non-optimized version.

9.3 Scalability Experiment

This experiment evaluates the performance of the proposed approach in large databases. As shown in Section 7.2, the optimization of the STP predicate is carried out without special index structures, which is practically preferred in the context of systems. It remains however questionable, how far are the traditional indexes (e.g. RTrees) effective for such a type of queries. This experiment tries to answer this question.

Obviously if all the lifted predicates within the STP predicate in a given query are not supported by the indexes in the database system, then one is out of luck, and the STP predicate will be evaluated for every tuple. Therefore, in this experiment, we compose the STP predicates by lifted predicates that are supported by index structures available in SECONDO.

9.3.1 Generating the Data

The data for this experiment is generated using the *BerlinMOD* benchmark [9]. It simulates an arbitrary number of cars moving in the city Berlin. The scenarios of the trips are quite realistic, simulating the trips to and from the work place, and the leasure time trips. The benchmark is downloadable from the SECONDO web site [4]. The trajectory data is generated by running SECONDO scripts. It is possible to control the number of cars, and the number of observation days by editing a configuration file.

For this experiment, we have generated the four relations described in Table 3. The table shows for every relation the number of cars/trajectories, the number of simulation days, the number of units of all trajectories, and the storage space of the relation. The number of units is analogous to the total number of observations of all cars, in the discrete sense. Note that in this moving objects model, the trajectories are continuous. That is, the locations of the cars between any two consecutive observations are linearly interpolated. The generation of the four relations using the *BerlinMOD* benchmark took about 5 days on the machine described in Section 9.

Table 3: The Database Relations Used In The Scalability Experiment

Relation Name	**Number of Cars**	**Duration**	**Number of Units**	**Size**
datascar50	50,000	1 day	64,331,426	9.1 GB
datascar100	100,000	1 day	128,437,840	18.2 GB
datascar200	200,000	1 day	256,373,737	36.3 GB
datascar300	300,000	1 day	384,923,972	54.5 GB

For each of the four relations, a spatial RTree index is derived for the *trip* attribute. The RTree contains the bounding boxes of the *units* of the *Trip* attribute, which are of type $\underline{upoint}$.

9.3.2 Generating the Queries

The BerlinMOD benchmark generates for every car up to five trips in a working day. Two of them go to and from the work place, and the other three trip are leasure time trips in the afternoon/evening. The leasure time destinations are randomly chosen from the *neighborhood* of the car's home location with a probability of 80%, and from the whole map with a probability of 20%. We use this information to design the experiment queries.

For each of the four relations in this experiment, a set of 10 queries is randomly generated. Each of the queries randomly picks a car, and retrieves its home location and three *locations* from its neighborhood, call them *atmmachine*, *supermarket*, and *bakery* for example. The query looks for the cars that made a leasure time trip starting from the location *home*, and passing by the locations *atmmachine*, *supermarket*, and *bakery* in order. Since the locations are chosen from the neighborhood of an existing car, there is some probability that the cars will fulfill the pattern. A sample query for the relation *datascar300* looks as follows:

```
SELECT count(*)
FROM datascar300 c
WHERE [ pattern([ c.trip = home as pred1,
                  c.trip = atmmachine as pred2,
                  c.trip = supermarket as pred3,
                  c.trip = bakery as pred4],
                [stconstraint("pred1", "pred2", later),
                 stconstraint("pred2", "pred3", later),
                 stconstraint("pred3", "pred4", later)])
      ]
```

where *home* is the home location of the car, and the = lifted predicate is fulfilled in the time instants/intervals when its two arguments have the same spatial coordinates. Ten such queries are randomly generated for every relation. The next subsection shows the average runtimes.

9.3.3 Results

In this experiment, we switch on the optimizer. Since the = lifted predicates in the queries belong to the *lifted spatial range* predicates, as shown in Table 2, the optimizer generates execution plans that use the RTree indexes, that are generated during the data generation. Figure 4 shows the average runtimes. These results conclude two points:

- Taking into consideration the large relation sizes as shown in Table 3, and the moderate machine specifications described in Section 9, the average runtimes are cheap regarding such complex query type. To be able to compare, we measured the average runtime of an optimized spatiotemporal range query on the 300,000 relation, and it shows 20 seconds. This is in comparison to an average of 28.6 seconds for the STP query. This confirms that the proposed optimization approach works fine without the need for specialized index structures.
- The runtime seems to scale linearly with the relation size. This is already expected since the STP predicate is applied to every tuple in the input (i.e. the tuples retrieved after the index access). Note that the BerlinMOD benchmark generates all the trips within the limited spatial space of the city Belrin. A larger number of cars in the simulation implies that the window queries on the RTree index yield more candidates.

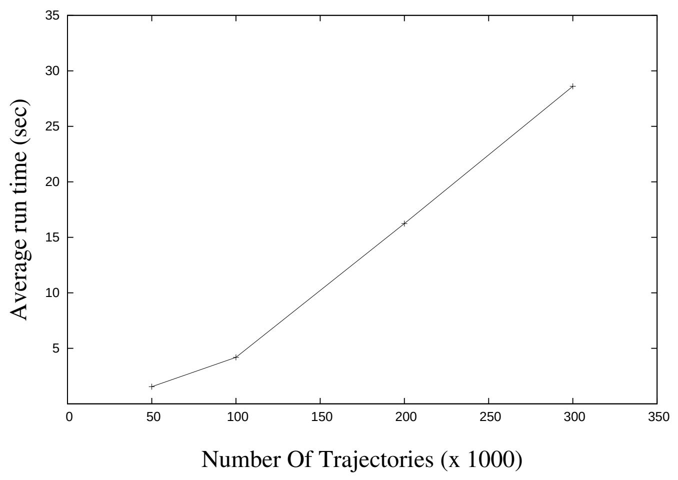


Figure 4: Scalability results

To sum things up, the scalability of the STP queries as proposed in this paper is affected by four parameters:

1. The number of lifted predicates in the STP predicate.
2. The number of the temporal constraints in the STP predicate.
3. The number of input tuples/trajectories.
4. The length of the trajectories in terms of number of units.

The scalability in terms of the first three parameters is evaluated already in the three experiments in this paper. The last parameter, the length of trajectories, affects the evaluation time of the STP predicate indirectly as it affects the evaluation time of the lifted predicates. This is because the lifted predicates are evaluated for the complete trajectory. When the trajectories are long (e.g. several weeks of observation time), the cost of evaluating the lifted predicates increases accordingly. The majority of them scale linearly with the number of units in the trajectory. More about the lifted predicate evaluation algorithms can be found in [8].

In the STP predicate, the temporal constraints impose a certain temporal order between the lifted predicates. While evaluating the STP predicate, one gets temporal information from the lifted predicates evaluated so far. A proper analysis of this information can identify parts of the trajectory that can be safely ignored while evaluating other lifted predicates. In future work we plan to study how to utilize this information. Roughly, one would need to redefine the lifted predicates, so that they process the trajectory parts upon request (e.g. in a stream fashion) rather than the whole trajectory.

10 Application Examples

To illustrate the expressive power of the proposed approach, we present in the following two subsections more examples for STP queries. Section 10.1 demonstrates a scenario called *Finding Ali*. It is about a kid called Ali, who moves on the street network of Cairo (the capital of Egypt). He makes several trips riding in several cars. We want to query for these cars using their movement profiles.

In Section 10.2, we demonstrate example queries that the reader can try himself/herself in SECONDO. The queries are based on the *berlintest* database, that is available with the SECONDO distribution. Unlike the first application, the queries are not linked to a single scenario. Hence we can demonstrate STP queries that involve moving points, moving regions, and many kinds of lifted operations.

10.1 Finding Ali

We assume that the road network of Cairo is observed for one month and that the complete trajectories of the cars are stored in the database. The queries assume the following schema:

- Car[PlatesNumber: $\underline{string}$, Trip: $\underline{mpoint}$] where Trip is the complete trajectory of the car for the whole observation period.
- Landmark[Name: $\underline{string}$, Type: $\underline{string}$, Location: $\underline{point}$]
- Heliopolis: A $\underline{region}$ object marking the boundary of the district *Heliopolis* where Ali lives.
- AliHome: A $\underline{point}$ object marking Ali's home.
- FamilyHome: A $\underline{point}$ object marking the house of the father's family.
- SportsClub: A $\underline{region}$ object marking the boundary of the sports club in which Ali is a member.

10.1.1 The Go-to-school Trips With the School Bus

The bus starts at the school at 6:00 am - 6:30 am, enters the district Heliopolis at 6:45 am - 7:00 am, stops near Ali's home, picks Ali, exits Heliopolis at 7:45 am - 8:00 am, then goes back to school.

This query can be written without a spatiotemporal pattern predicate. The spatiotemporal window of every predicate is known. It can be expressed as a conjunction of 5 spatiotemporal range predicates (Bus inside School at the time interval [6:00, 6:30] AND Bus inside Heliopolis at the time interval [6:45, 7] ...). We include this as an example of spatiotemporal pattern queries that can be expressed without STP predicates.

10.1.2 The Evening Trips With Grandfather

Starting from Ali's home, the grandfather drives Ali to the sports club. They stop at the sports club for at least two hours. After the club they go by car to buy some bread, then back home.

```
SELECT c.PlatesNumber
FROM Car c, Landmark l
WHERE  l.Type like("%Bakery%") and
  patternex([distance(c.Trip, AliHome) < 20.0 as AtHome,
    c.Trip inside SportsClub as AtClub,
    distance(c.Trip, l.Location) < 20.0 as AtBakery,
    distance(c.Trip, AliHome) < 20.0 as BackHome],
  [AtHome later AtClub,
    AtClub later AtBakery,
    AtBakery later BackHome],
  end("AtClub") - start("AtClub") >= 2.0 and
  daypart(AtHome) = daypart(BackHome))
```

In this query, the extended STP predicate is used to state that they stayed at least two hours in the sports club and that the whole pattern occurred in one day. Another note is that the query uses the predicate `distance(c.Trip, AliHome) < 20.0` twice with two different aliases. The two aliases are needed to write the constraints. It is the responsability of the query optimizer to detect this common predicate (i.e. using common sub-expression optimization techniques) and evaluate it only once.

10.1.3 The Weekend Trips With Mother

The mother starts from Ali's home, drives only in main roads, stops near a shopping mall for at most 4 hours then back home. The trip to the mall takes more than 1.5 times the estimated time because the mother uses only main roads. In Cairo it is easier to drive in main roads but they have high traffic.

```
SELECT c.PlatesNumber
FROM   Car c, Landmark l
WHERE  l.Type like("%Mall%") and
  patternex([distance(c.Trip, AliHome) < 20.0 as AtHome,
    distance(c.Trip, l.Location) < 40.0 as AtMall,
    distance(c.Trip, AliHome) < 20.0 as BackHome],
  [AtHome later AtMall,
    AtMall later BackHome],
  end("AtMall") -  start("AtMall") <= 4.0 and
  (start("AtMall") - end("AtHome") >
    1.5 * EstimatedDriveTime(l.location, AliHome) ))
```

where we assume for simplicity that *EstimatedDriveTime* is a function that computes the normal period that a drive between two places takes. It may do so by finding the shortest path and multiply by the average driving speed.

10.2 The Berlintest Example

In this example, we use the database *berlintest*, more specifically, the *Trains* relation and three newly added relations with the following schemas:

SnowStorms[Serial: $\underline{int}$, Storm: $\underline{mregion}$]

TrainsMeet[Line: $\underline{int}$, Uptrip: $\underline{mpoint}$, Downtrip: $\underline{mpoint}$, Stations: $\underline{points}$]

TrainsDelay[Id: $\underline{int}$, Line: $\underline{int}$, Actual: $\underline{mpoint}$, Schedule: $\underline{mpoint}$]

The *SnowStorms* relation contains 72 tuples, each of which contains a moving region, representing a snow storm that moves over Berlin. The *TrainsMeet* relation is generated from the *Trains* relation. The tuples contain all possible combinations of two trains that belong to the same line and move in opposite directions. The *Stations* attribute represents the train stations of the associated line. The *TrainsDelay* relation is also generated from the *Trains* relation. Each tuple contains the original *Trip* attribute (renamed into *Schedule*), and a delayed copy of it with delays of around 30 minutes. The scripts for creating the three relation and for executing the example queries are available for download as will be explained in Appendix D.

Table 4 lists the lifted operations used within the queries. We have designed the queries so that they illustrate the expressive power of our approach by using various lifted operations to compose complex pattern queries. The table shows only the operator signatures that are used in the queries. The complete list of valid signatures is in [19].

Table 4: Lifted Operations

Operation	**Signature**	**Type**	**Meaning**
at	$\underline{mregion} \times \underline{point} \rightarrow \underline{mpoint}$	topological operation	computes a moving point that exist whenever the point argument is inside the moving region argument.
isempty	$\underline{mpoint} \rightarrow \underline{mbool}$	set operation	true whenever the argument is defined.
not	$\underline{mbool} \rightarrow \underline{mbool}$	boolean operation	logical negation.
rough_center	$\underline{mregion} \rightarrow \underline{mpoint}$	aggregation	aggregates the moving region into a moving point that represents its center of gravity.
speed	$\underline{mpoint} \rightarrow \underline{mreal}$	metric property	the metric speed of the moving point.
distancetraversed	$\underline{mpoint} \rightarrow \underline{mreal}$	metric property	the distance that the moving point traversed since the start of its definition time.
area	$\underline{mregion} \rightarrow \underline{mreal}$	metric property	the area of the moving region.
intersection	$\underline{mpoint} \times \underline{mpoint} \rightarrow \underline{mpoint}$	set operation	computes the common parts of the two arguments.
inside	$\underline{mpoint} \times \underline{mregion} \rightarrow \underline{mbool}$ $\underline{mpoint} \times \underline{points} \rightarrow \underline{mbool}$	spatial range predicate	true whenever the $\underline{mpoint}$ is contained in the $\underline{mregion}$, or passes some of the $\underline{points}$.
delay	$\underline{mpoint} \times \underline{mpoint} \rightarrow \underline{mreal}$	metric operation	considers the first argument *actual*, and the second *schedule movement* and computes the delay of the actual movement in seconds.
=	$\underline{mpoint} \times \underline{point} \rightarrow \underline{mbool}$	spatial range predicate	true whenever the moving point passes the point.
xangle	$\underline{mpoint} \rightarrow \underline{mreal}$	direction	the angle (in degrees) between x-axis and the tangent of the moving point.
and	$\underline{mbool} \times \underline{mbool} \rightarrow \underline{mbool}$	boolean operation	logical and.
<, <=, >, >=	$\underline{mreal} \times \underline{real} \rightarrow \underline{mbool}$	left/right range predicate	true in the time intervals during which the comparison holds.

10.2.1 Find the snow storms that passed over the train station *mehringdamm* with speed greater than 40 km/h.

```
SELECT *
FROM   snowstorms
WHERE  pattern([not(isempty(storm at mehringdamm)) as pred1,
         speed(rough_center(storm)) > 40.0 as pred2],
          [stconstraint("pred1","pred2", together)])
```

where *together* is a vector temporal connector that yields true if the two predicates happen simultaneously.

10.2.2 Find the snow storms that could increase their area over 1/4 square km during the first traversed 5 km.

```
SELECT *
FROM   snowstorms
WHERE  pattern(
       [distancetraversed(rough_center(storm)) <= 5000.0 as pred1,
         area(storm) > 250000.0 as pred2],
       [stconstraint("pred1","pred2", meanwhile)])
```

10.2.3 Find the trains whose up and down trips meet inside one of the train stations.

```
SELECT   *
FROM     trainsmeet
WHERE    pattern(
         [not(isempty(intersection(uptrip, downtrip))) as pred1,
           uptrip inside stations as pred2 ],
         [stconstraint("pred1","pred2", together)])
ORDERBY  line
```

10.2.4 Find the trains that encountered a delay of more than 30 minutes after passing through the snow storm *msnow*.

```
SELECT *
FROM   trainsdelay
WHERE  pattern([not(delay(actual, schedule) > 1800.0) as pred1,
         actual inside msnow as pred2,
         delay(actual, schedule) > 1800.0 as pred3 ],
       [stconstraint("pred1", "pred2", vec("abab", "aba.b", "abba")),
       stconstraint("pred2", "pred3",
          vec("abab", "aba.b", "abba", "aa.bb", "aabb"))])
```

10.2.5 Find the trains that are always heading north-west after passing *mehringdamm*.

```
SELECT *
FROM   trains
WHERE  patternex([trip = mehringdamm as pred1,
         ndefunit(((xangle(trip) >= 90.0) and
           (xangle(trip) <=180.0)), int2bool(1)) as pred2],
       [stconstraint("pred1","pred2",then)],
       (((start("pred2")- end("pred1")) < create_duration(0, 120000))
       and
       ((inst(final(trip)) - end("pred2")) < create_duration(0, 15000))))
```

where we use the *ndefunit* operator in this query to replace the undefined periods within the <u>*mbool*</u> by *true* units. This is because the *xangle* [2] operator yields undefined during the train stops in the stations. In other words, *pred2* is true whenever the train is not heading other than north-west. The query restricts the results to the trains which started heading north at most 2 minutes after passing *mehringdamm* and

[2]The *xangle* operator is a corrected copy of the SECONDO *mdirection* operator. It is presented only for the sake of this example. In the SECONDO versions newer than 2.9.1, the *mdirection* operator works fine.

remained so till at least 15 seconds before the end of the trip. These time margins are used to cut out small noisy parts in the data, so that the query yields results.

11 System Use and Experimental Repeatability

The implementation of the described approach is made available as a Plugin for the SECONDO system. It can be downloaded from the Plugin web site [1]. The *User Manual* (also available on the Plugin we site) describes how to install and run the Plugin. We have also made available the scripts for running the first and the second experiments in this paper, and the *Berlintest* application example, so that the results are repeatable. There are no scripts here for the third experiments. For interested readers, please refer to the *BerlinMOD* benchmark [9] to generate the test data, then use the queries as described in Section 9.3.

Before running the scripts of the experiments, you need to install:

1. The SECONDO system version 2.9.1 or later [3]. A brief installation guide is given in the *Plugin User Manual* on [1], and a detailed guide is given in the SECONDO *User Manual* [3].
2. The Spatiotemporal Pattern Queries Plugin (STPatterns) as described in [1].

11.1 Repeating the First Experiment

During the installation of the STPattern Plugin, two files are copied to the SECONDO bin directory \$SECONDO_BUILD_DIR/ bin. These two files *Expr1Script.sec* and *STPQExpr1Query.csv* (described in Appendix A) automate the repeatability of the first experiment in this paper. The experiment can then be run as follows:

1. Run SecondoTTYNT (i.e. in a shell, go to \$SECONDO_BUILD_DIR/bin and write `SecondoTTYNT`).
2. Make sure that the berlintest database is restored (i.e. at the SECONDO prompt, write `list databases` and make sure that berlintest database is in the list). Otherwise, restore it by writing

   ```
   restore database berlintest from berlintest
   ```

   at the SECONDO prompt (press <return> twice).
3. Execute the script by writing `@Expr1Script.sec` at the SECONDO prompt. The script creates the required database objects and executes the experiment queries. This may take half an hour depending on your machine.

Executing the script creates a SECONDO relation *STPQExpr1Result* in the *berlintest* database, which stores the experimental results. Its schema is shown in Table 5.

The experimental results are also saved to a comma separated file *STPQExpr1Result.csv* in the SECONDO bin directory. The file has a similar structure as the table *STPQExpr1Result*.

11.2 Repeating the Second Experiment

Repeating the second experiments is also automated by script files that are copied to the SECONDO directories during the installation of the STPattern Plugin. For the second experiment, two script files are used; the *\$SECONDO_BUILD_DIR/ bin/ Expr2Script.sec* file creates the necessary database objects, and the *\$SECONDO_BUILD_DIR/ Optimizer/ expr2Queries.pl* executes the queries. The *Expr2Script.sec* file is described in Appendix B, and the *expr2Queries.pl* in Appendix C. The experiment is repeated as follows:

[3]Since our optimizer extension wraps around the standard optimizer implementation, you may get different optimization results in later SECONDO versions. The described results in this paper are obtained from version 2.9.1

Table 5: The schema of the STPQExpr1Result relation

Attribute	**Meaning**	**Example**
no	A serial number for the query.	0
queryText	The query text.	`thousand feed filter [.stpattern[ a:passmbool(mb10), b:passmbool(mb30); stconstraint("a", "b", vec("aa.b.b"))]] count`
numPreds	The number of the lifted predicates in the STP predicate.	2
numConstraints	The number of the constraints in the STP predicate.	1
ElapsedTimeReal	The measured response time, in seconds, for this query.	0.171932
ElapsedTimeCPU	The measured CPU time, in seconds, for this query	0.16

1. Run SecondoTTYNT.
2. Make sure that the berlintest database is restored, otherwise, restore it.
3. Execute the *Expr2Script.sec* by writing `@Expr2Script.sec` at the SECONDO prompt. This creates the necessary database objects.
4. Quit SecondoTTYNT (i.e. write `quit` at the SECONDO prompt), go to the SECONDO optimizer folder *$SECONDO_BUILD_DIR/ Optimizer* and write `SecondoPL`. This starts the SECONDO optimizer user interface in the single user mode.
5. Write `consult(expr2Queries).` to let Prolog interpret the script file *expr2Queries.pl*.
6. Open the *berlintest* database (i.e. write `open database berlintest.`).
7. Write `runSTPQExpr2DisableOptimization.` to run the queries without enabling the optimization of the STP predicate, or `runSTPQExpr2EnableOptimization.` to run the queries with the optimization of the STP predicate being enabled. This can take more than an hour.

The results are saved to the comma separated files *Expr2StatsDO.csv* and *Expr2QueriesDO.csv* in the SECONDO optimizer folder if the STP predicate optimization is disabled. If it is enabled, the results are saved to the files *Expr2StatsEO.csv* and *Expr2QueriesEO.csv*.

The files *Expr2StatsDO.csv* and *Expr2StatsEO.csv* show the run times. They include the columns described in Table 6.

Table 6: The schemas of the Expr2StatsDO.csv and Expr2StatsEO.csv files

Attribute	**Meaning**	**Example**
NumberOfPredicates	The number of the lifted predicates in the STP predicate.	2
NumberOfConstraints	The number of the constraints in the STP predicate.	1
Serial	A serial for the query in the range [0,9]. The serial is repeated with every experimental setup	1
ExecTime	The measured response time, in milliseconds, for this query.	443

The files *Expr2QueriesDO.csv* and *Expr2QueriesEO.csv* have a similar structure. They exclude the *ExecTime* attribute and have two more attributes; the *SQL* attribute which stores the SQL-like query, and the *ExecutablePlan* which stores the execution plan generated by the Optimizer.

12 Conclusions

We propose a novel approach for spatiotemporal pattern queries. It combines efficiency, expressiveness and a clean concept. It builds on other moving objects database concepts. Therefore, it is convenient in the context of spatiotemporal DBMSs. Unlike the previous approaches, it is integrated with query optimizers. We also propose an algorithm for evaluating the constraint satisfaction problems, that is customized to fit the efficient evaluation of the spatiotemporal pattern predicates. In the paper, we demonstrate two application examples to emphasize the expressive power of our approach. Our work is completely implemented in the SECONDO platform. The implementation and the scripts for experimental repeatability are available on the Web. The experimental evaluation shows that the run times are reasonable. As future work, we intend to revisit the definition of the lifted predicates, and extend them to process only the parts of the trajectories that are candidates for a solution of the STP predicate. This will allow for efficiently reporting patterns in long trajectories.

References

[1] SECONDO plugins.
http://dna.fernuni-hagen.de/secondo.html/start_content_plugins.html.

[2] SECONDO programmer's guide.
http://dna.fernuni-hagen.de/secondo.html/files/programmersguide.pdf.

[3] SECONDO user manual.
http://dna.fernuni-hagen.de/secondo.html/files/secondomanual.pdf.

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

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

[6] Luis Otavio Alvares, Vania Bogorny, Bart Kuijpers, Jose Antonio Fernandes de Macedo, Bart Moelans, and Alejandro Vaisman. A model for enriching trajectories with semantic geographical information. In *GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[7] Christian Bessiere. *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[8] José Antonio Cotelo Lema, Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. Algorithms for moving objects databases. *Comput. J.*, 46(6):680–712, 2003.

[9] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. BerlinMOD: a benchmark for moving object databases. *The VLDB Journal*, 18(6):1335–1368, 2009.

[10] Martin Erwig. *Toward Spatiotemporal Patterns, Spatio-Temporal Databases (ed. Caluwe, De)*, chapter 2, pages 29–54. Springer-Verlag New York, Inc., 2004.

[11] Martin Erwig and Markus Schneider. Developments in spatio-temporal query languages. In *DEXA '99: Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, page 441, Washington, DC, USA, 1999. IEEE Computer Society.

[12] Martin Erwig and Markus Schneider. Spatio-temporal predicates. *IEEE Trans. on Knowl. and Data Eng.*, 14(4):881–901, 2002.

[13] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus 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.

[14] Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. Algorithms for nearest neighbor search on moving object trajectories. *Geoinformatica*, 11(2):159–193, 2007.

[15] Joachim Gudmundsson, Marc van Kreveld, and Bettina 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.

[16] Ralf Hartmut Güting, Victor Almeida, Dirk Ansorge, Thomas Behr, Zhiming Ding, Thomas Höse, Frank Hoffmann, Markus Spiekermann, and Ulrich 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.

[17] Ralf Hartmut Güting, Thomas Behr, Victor Almeida, Zhiming Ding, Frank Hoffmann, and Markus Spiekermann. SECONDO: An extensible DBMS architecture and prototype. Technical Report Informatik-Report 313, FernUniversität Hagen, March 2004.

[18] Ralf Hartmut Güting, Thomas Behr, and Jianqiu Xu. Efficient *k*-nearest neighbor search on moving object trajectories. In *The VLDB Journal, Online First*, 2010.

[19] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

[20] Marios Hadjieleftheriou, George Kollios, Petko Bakalov, and Vassilis J. Tsotras. Complex spatio-temporal pattern queries. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 877–888. VLDB Endowment, 2005.

[21] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[22] Cédric Mouza and Philippe Rigaux. Mobility patterns. *Geoinformatica*, 9(4):297–319, 2005.

[23] Nikos Pelekis, Ioannis Kopanakis, Gerasimos Marketos, Irene Ntoutsi, Gennady Andrienko, and Yannis Theodoridis. Similarity search in trajectory databases. In *TIME '07: Proceedings of the 14th International Symposium on Temporal Representation and Reasoning*, pages 129–140, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Markus Schneider. Evaluation of spatio-temporal predicates on moving objects. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 516–517, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: Issues and solutions. In *SSDBM'98: 10th International Conference on Scientific and Statistical Database Management*, pages 111–122, 1998.

A The Expr1Script.sec File

This is a commented version of the *Expr1Script.sec* script.

The script runs the first experiment with minimal user interaction. The experiment, as described in Section 9.1, is intended to evaluate the execution overhead of the STP predicates. This script first creates the required database objects, then executes the queries and logs the run times.

```
close database;
open database berlintest;

let mb1 = randommbool(now());
 ⋮
let mb30 = randommbool(now());
```

The commands open the database *berlintest* and creates 30 random <u>*mbool*</u> objects with the names *mb1... mb30*. These objects are needed for the queries. The *randommbool* operator works as described in Section 9.1.1.

```
let later = vec("aabb", "a.abb", "aab.b", "a.ab.b");
let follows = vec("aa.bb", "a.a.bb", "aa.b.b", "a.a.b.b");
let immediately = vec("a.bab", "a.bba", ...
let meanwhile = vec(  ...
let then = vec( ...
```

The five vector temporal connectors are used in the queries as examples for vector temporal connectors. They are used together with the 26 simple temporal connectors to generate the queries.

```
let STPQExpr1Query=
  [const rel(tuple([no:int, queryText: text,
    numPreds: int, numConstraints: int])) value ()]
  csvimport['STPQExpr1Query.csv', 0, "", "$"] consume;
```

The query imports the experiment queries from the comma separated file *STPQExpr1Query.csv* and stores them in a SECONDO relation called *STPQExpr1Query*. The [*const* . *value* .] operator tells the *cvsimport* operator the schema of the relation, which is shown in Table 7.

Table 7: The schemas of the STPQExpr1Query.csv file and the STPQExpr1Query table

Attribute	**Meaning**
no	A serial for the query in the range [0, 4899].
queryText	The query statement written in SECONDO executable language.
numPreds	The number of the lifted predicates in the STP predicate.
numConstraints	The number of the constraints in the STP predicate.

The file contains 4900 queries that were randomly generated as described in Section 9.1.2. The queries represent 49 experimental settings, each of which have 100 queries. The following query executes them and logs the results in the relation *STPQExpr1Result*:

```
let STPQExpr1Result =
  STPQExpr1Query feed
  loopjoin[fun(queryTuple: TUPLE)
    evaluate(attr(queryTuple, queryText))
  project[ElapsedTimeReal, ElapsedTimeCPU]]
  consume;
```

This query can take half an hour depending on your machine. You can query the results relation in any of the SECONDO user interfaces [3] and create aggregations for the charts. Additionally, the following query exports the relation to the comma separated file *STPQExpr1Result.csv* in the SECONDO bin directory.

```
query STPQExpr1Result feed
  projectextend[; Serial: .no,
    NumberOfPredicates: .numPreds,
    NumberOfConstraints: .numConstraints,
    ResponseTime: .ElapsedTimeReal,
    CPUTime: .ElapsedTimeCPU]
csvexport['STPQExpr1Result.csv', FALSE, TRUE]
count
```

NOTE: We encourage the reader to get information about the SECONDO operators by using the built-in operator descriptions. For example, to get help on the operator `csvimport`, write the following query at the SECONDO prompt:

```
query SEC2OPERATORINFO feed
  filter[.Name contains "csvimport"]
consume
```

B The Expr2Script.sec File

This is a commented version for the *Expr2Script.sec* script.
The script is used to generate the data required for running the second experiment in this paper without executing the queries. The queries need to be executed in the *SecondoPL* environment afterwards.

```
close database;
open database berlintest;

let RestaurantsNumbered =
  Restaurants feed addcounter[no, 1] head[300] consume;
let point1 =
  RestaurantsNumbered feed filter[.no = 1] extract[geoData];
 ⋮
let point300 =
  RestaurantsNumbered feed filter[.no = 300] extract[geoData];
delete RestaurantsNumbered;
```

First, the commands open the database *berlintest*. The geometries of the first 300 restaurants in the *Restaurants* table are then copied to point objects (point1... point300) to be used in the queries.

```
let later = vec("aabb", "a.abb", "aab.b", "a.ab.b");
let follows = vec("aa.bb", "a.a.bb", "aa.b.b", "a.a.b.b");
let immediately = vec("a.bab", "a.bba", ...
let meanwhile = vec(  ...
let then = vec( ...
```

The five vector temporal connectors, that are also created in *Expr1Script.sec*, are included here so that the two experiments can be run independently.

```
let Trains20 = thousand feed head[20] Trains feed product consume;
```

This query creates the *Trains20* relation by replicating the tuples of the *Trains* relation 20 times. In the following query, we create an index on the *Trains20* relation to test the proposed STP predicate optimization. The index is a spatial R-tree on the units of the *Trip* attribute. Instead of indexing the complete movement, the index is built on the units (i.e. a bounding box is computed for every unit in the Trip). This is done so that the bounding boxes better approximate the moving point.

```
let Trains20_Trip_sptuni =
  Trains20 feed
    projectextend[Trip; TID: tupleid(.)]
    projectextendstream[TID; MBR: units(.Trip)
      use[fun(U: upoint) bbox2d(U) ]]
    sortby[MBR asc]
  bulkloadrtree[MBR];
```

C The expr2Queries.pl File

This Prolog file is used to run the queries of the second experiment and log the execution times. It defines four prolog predicates:

1. runSTPQExpr2DisableOptimization/0: switches off STP predicate optimization by setting the optimizer options, and executes the queries.

2. runSTPQExpr2EnableOptimization/0: switches on STP predicate optimization, and executes the queries.

3. executeSQL/4: helper predicate for executing queries.

4. runSTPQExpr2/4: the facts table that stores the queries. The file contains 490 such facts, 10 queries for each of the 49 experimental settings. The queries are randomly generated as described in Section 9.2.2. For every query, the fact also stores its serial, number of lifted predicates, and number of constraints.

D Running the Berlintest Application Example

To execute the queries in the berlintest example, you need first to run the script *BerlintestScript.sec* from the SecondoTTYNT prompt. The script is installed within the STPattern Plugin. You also need to have the berlintest database restored in your system. The script file creates the required database objects but it doesn't execute the queries. It first defines some temporal connectors:

```
close database;
open database berlintest;
let later= vec("aabb", "a.abb", "aab.b", "a.ab.b");
let follows= vec(...
let immediately= vec(...
let meanwhile= vec(...
let then= vec(...
let together= vec(...
```

Then it restores the *SnowStorms* relation from the *SnowStorms* file in the SECONDO/bin directory, which is installed with the Plugin.

```
restore SnowStorms from SnowStorms;
```

The following command creates the relation *TrainsMeet*, that is used in the example in Section 10.2.3. Every tuple in the relation is a different combination of an up train, down train of the same line, and the stations where the train line stops.

```
let TrainsMeet =
  Trains feedproject[Line, Trip, Up] {t2}  filter[.Up_t2 = FALSE]
  Trains feedproject[Line, Trip, Up] {t1}  filter[.Up_t1 = TRUE]
  hashjoin[Line_t2 , Line_t1 , 99997]
  extend[Line: .Line_t1, Uptrip: .Trip_t1, Downtrip: .Trip_t2,
    Stations: ((breakpoints(.Trip_t1, create_duration(0,5000) )
      union val(initial(.Trip_t1)))
      union val(final(.Trip_t1)))]
  project[Line, Uptrip, Downtrip, Stations]
  consume;
```

Next we create the relation *TrainsDelay*, used in the example in Section 10.2.4. Every tuple has a *schedule* and an *actual* moving point. The *schedule* movement is a copy from the *Trip* attribute in the *Trains* relation. The actual movement should have delays of about half an hour. We shift the *Trip* 1795 seconds forward, and apply a random positive or negative delay up to 10 seconds to the result. This creates actual movements with random delays between 29:45 and 30:05 minutes.

```
let TrainsDelay=
  Trains feed
  extend[Schedule: .Trip,
    Actual: randomdelay(
      .Trip translate[create_duration(0, 1795000) , 0.0, 0.0],
      create_duration(0,10000) ) ]
  project[Id, Line, Actual, Schedule]
  consume;
```

After running the *BerlintestScript.sec* script, use the *Javagui* to execute the queries. It is the graphical user interface for SECONDO. To launch it:

1. Start the SECONDO kernel in server mode, the optimizer server, and the GUI:
   In a new shell, go to $SECONDO_BUILD_DIR/bin, and type `SecondoMonitor -s`.
   In a new shell, go to $SECONDO_BUILD_DIR/Optimizer, and type `StartOptServer`.
   In a new shell, go to $SECONDO_BUILD_DIR/Javagui, and type `sgui`. The Javagui will start and connect to both the kernel and the optimization server.

2. Open the database. In the Javagui type:
   `open database berlintest`.

3. Set the optimizer options. The SECONDO optimizer maintains a list of options that controls the optimization. The examples in this paper require the options *improvedcosts*, *determinePredSig*, *autoSamples*, *rewriteInference*, *rtreeIndexRules*, and *autosave*. To set each of these options, type in the Javagui:
   `optimizer setOption(`*`option`*`)`

4. View the underlying network. Type:
   `select * from ubahn` to display the underground trains network.
   `select * from trains` to display the moving trains. Use the slider to view the results.
   Select the last query in the top-right panel and press hide to hide the trains.
   `select * from snowstorms` to display the moving snow storms.
   hide the snow storms.

5. Type the example queries as in Section 10.2, and make sure to type everything in lower case.