A Point-based Temporal Extension of SQL

David Toman*

Department of Computer Science, University of Toronto
Toronto, Ontario M5S 1A4, Canada
david@cs.toronto.edu

Abstract. We propose a new approach to temporal extensions of SQL. Unlike the current proposals, e.g., SQL/Temporal, we use *point-based* references to time as the basis of our approach. The proposed language—SQL/TP—extends the syntax and semantics of SQL/92 in a very natural way: by adding a single data type to represent a linearly ordered universe of *individual time instants*. Such an extension allows the users to write temporal queries in customary fashion and *vastly* simplifies the semantics of the proposed language: we merely use the familiar SQL semantics. In this way SQL/TP also fixes many problems present in the semantics of the temporal query languages based on explicit interval-valued temporal attributes. In addition, we propose an *efficient* query evaluation procedure over a compact *interval-based* encoding of temporal relations. The algorithm is based on a *sophisticated compilation* technique that translates SQL/TP queries to SQL/92. In this way existing database systems can be used for managing temporal data. We substantiate this claim by proposing an experimental version of a SQL/TP compiler to serve as a front-end for DB2[1] [13].

1 Introduction

A large amount of database research is directed towards the limitations of the classical relational model and on ways to overcome these limitations. The fruits of this research are slowly finding their place in the mainstream commercial systems, e.g., through new SQL standards. The major developments in this area are, e.g., the introduction of deductive features or the object-relational extensions of the relational model. While the first extension is aimed on overcoming limitations in expressive power of relational queries, the second approach is directed towards handling *interpreted* data (rather than mere uninterpreted constants). In this paper we propose a different approach to introducing *interpreted data* into the relational model. We also propose *a sophisticated compilation technique* that allows us to handle such an extension in a standard relational system. While we are mostly concerned with adding *temporal capabilities*, the proposed technique can be extended to other interpreted domains of data, e.g., spatial data. We address the following issues:

- We show both theoretical and practical reasons, why the current proposals of temporal extensions of SQL are inadequate.
- We show that there is a simpler and more natural temporal extension of SQL based on a point-based view of time.

* This research was supported by a NATO/NSERC PDF fellowship, 1996-98.

[1] DB2 is a trademark of IBM Corp.

– We propose a query execution model for our language that allows *efficient* query evaluation over temporal databases encoded using interval-based timestamps.

The three main technical contributions of this paper are: (1) the definition of a *representation-independent* temporal extension of SQL: we decouple the syntax of the language from the underlying data representation while preserving SQL's semantics: we support both set- and duplicate-based semantics including aggregation, (2) a *query compilation technique* for such an extension that allows SQL/TP queries to be efficiently evaluated using a *standard* RDBMS, and (3) the definition of nouveau *normalization* technique that facilitates evaluation of temporal queries over an interval-based encoding of timestamps. We would also like to note at this point that a naive direct compilation technique *fails* to achieve efficient query evaluation.

The rest of the paper is organized as follows: Section 2 explains the shortcomings of the current proposals of temporal extensions of SQL and sketches out the solution. Section 3 introduces the temporal data model: the abstract and concrete (interval-based) temporal databases (following the terminology introduced in [9]). Section 4 defines the syntax and semantics of SQL/TP and gives examples of temporal queries. Section 5 gives a sketch of the proposed compilation technique. The paper is concluded with several open questions and directions of future research.

2 Why another temporal extension of SQL?

The last decade of research in temporal databases has led to the development of several temporal query languages based on extensions of existing relational languages, e.g., TQUEL [16] or various temporal extensions of SQL, most prominent of which are TSQL2 [17] and its variants: ATSQL2 [5] and the current proposal of temporal extension of SQL3 to the ISO and ANSI standardization committees—SQL/Temporal [18].

2.1 Current Proposals

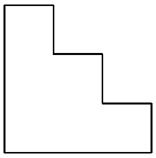
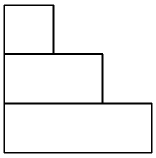
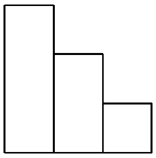
All the current proposals recognize that timestamping ordinary tuples with single time instants leads to enormous space requirements: a tuple would have to be *repeated* for every time instant at which the fact represented by the tuple holds. Instead, compact encodings of sets of time instants (often called *periods* of validity) associated with a tuple are *encoded* using intervals [16, 18], *bitemporal elements* [4, 14], or other fixed-dimensional products of intervals (hyperrectangles). The chosen encoding then provides a domain of values for temporal attributes.

However, in all the above approaches, the explicit access to the interval-based encoding of timestamps in temporal relations leads to a *tension* between the syntax and the intended semantics of the languages, specifically:

– References to time are realized using temporal attributes *explicitly referring to intervals* (or some other particular encoding of timestamps).
– The data model and the semantics are *point-based* [4, 9]: the intervals are used merely as compact descriptions of large sets of time instants (even in cases when one of the various duplicate semantics is used).

This conflict leads to *many unpleasant surprises*, especially when multiple temporal dimensions are needed to evaluate a given query (e.g., when the query simultaneously references two distinct points in time in an essential way). Most importantly, it is easy to show examples of queries whose *answers depend on the choice of the particular encoding rather than on the underlying meaning*, cf. Example 2.1 below. In addition, it is extremely hard to avoid such a behavior in an elegant way, and the actual semantics of these languages tend to be very cumbersome (if they exist at all). In many cases uniqueness of answers can only be guaranteed by operational means, e.g., by prescribing a particular evaluation order. Moreover, duplicate semantics and results of *aggregation* operations in such languages also inherently depend on the chosen encoding. Consider the following situation:

Example 2.1 Let D be a temporal relation (or an answer to a temporal query) that represents the region in the figures below.

(1)  (2)  (3) 

It is important to understand that all the figures *represents the same relation*. However, it is also clear that we can distinguish (2) and (3) using a first-order query in, e.g., SQL/Temporal. We call such queries *representation dependent*. Moreover, even very simple queries, e.g., counting the number of regions along the axes, give different results depending on the particular representation.

In the rest of this section we argue that the situation in Example 2.1 naturally arises during evaluation of first-order temporal queries.

First, we argue that a single temporal dimension is not sufficient to formulate general temporal queries. Consider the query "are there two distinct time instants when a given relation contains exactly the same tuples?" [1] and [20] have independently shown that this query cannot be formulated in first-order temporal logic. A direct corollary of this result is that this query cannot be expressed in *any* single-dimensional temporal relational algebra[2]. Moreover, [20] shows that to express all first-order queries the number of temporal dimensions cannot be bounded by any constant. Therefore, multiple temporal dimensions cannot be avoided during the bottom-up evaluation of temporal queries even if the final result is a single-dimensional temporal relation or boolean [20]. This fact, combined with the use of explicit interval-valued temporal attributes, leads directly to situations similar to Example 2.1.

Now it is easy to see why the coalescing-based approaches fail to guarantee representation independence: To guarantee fixed size of tuples in a temporal relation the region (1) in Example 2.1 has to be represented by a finite union of rectangles, e.g., using the representation (2) or (3) above. While both (2) and (3) are *coalesced*, they can still be distinguished by a first-order query with interval-valued temporal attributes. Moreover, this problem cannot be avoided using a different normal form as there is no unique coalescing-based normal form for dimension higher than one (for detailed discussion of coalescing see [6]).

In addition, in many cases the user has no control over the representation of the intermediate results since the coalescing is performed by the system implic-

[2] A relational algebra over the universe of *single-dimensional* temporal relations.

itly. While coalescing in a single-dimensional system facilitates representation independent formulation of queries, with two or more temporal dimensions it leads to serious problems: the user has no knowledge if the region (1) in Example 2.1 is going to be represented as (2) or (3) during the query evaluation. However, results of queries depend on this information.

2.2 Our Proposal

The above problems are inherent to most of the temporal query languages with temporal attributes ranging over intervals. Therefore we follow a different path to avoid all of the above problems: we let the temporal attributes in our language range over *single time instants*. In this way our approach separates an abstract query language—SQL/TP—defined over a clean model of point-based linearly ordered time from the concrete interval-based encoding of timestamps which is *hidden* from the user. The approach is based on several recent results in the area of temporal and constraint query languages [1, 15, 19, 20]. In addition, a we define a meaningful approach to duplicate semantics and aggregation that is independent of the particular encoding (using the results in [10, 11]).

While we mostly concentrate on evaluation of temporal queries over the interval-based encoding of time, conforming to the above principles allows us to use different encoding for sets of time instants, e.g., the *linear repeating points* [22] for periodic events, *without* the need for new syntax and semantics. In addition our proposal meets the following requirements:

- SQL/TP can be efficiently implemented on top of an interval-based representation of temporal databases: the performance of a SQL/TP system should be comparable to performance of SQL/Temporal based DBMS on a vast majority of representation independent queries.
- SQL/TP statements can be compiled to standard SQL/92[3] [12]; the translated queries can be evaluated using an off-shelf database system. This way we can build a SQL/TP front-end to an *existing* RDBMS and provide temporal capabilities *without* modifying the underlying database system itself.
- SQL/TP can express all representation independent SQL/Temporal queries. Moreover, SQL/TP is complete in the sense of [8]. The results in [1, 20] show that this is not the case for any of the temporal query languages based on a fixed-dimensional temporal relational algebra, e.g., [7]; this issue is not clear for TSQL2-derived languages [5, 17, 18].
- Our language can be easily extended to support the *migration requirements* [18] that require several levels of *temporal upward compatibility* with SQL. While SQL/TP itself does not literally follow all the requirements, the compatibility can be easily achieved using a very simple syntactic manipulation of the source queries and adding tags to distinguish the particular compatibility modes.

Before we start the technical part of the paper, we would like to reiterate (to avoid any misunderstanding) that we are interested in intervals *as a physical encoding of sets on time instants*. This is very different from the approaches

[3] Other relational languages can be used as well, provided they have sufficient expressive power.

taken in the various *interval logics* [2], where intervals represent *points* in a two-dimensional (half-)space. However, due to the natural multidimensionality of SQL/TP, we can represent the *true intervals* using pairs temporal attributes.

3 The Data Model for Temporal Databases

We start with the definition of the underlying data model: the domain of time is viewed as a discrete[4] countably infinite linearly ordered set without endpoints (e.g., the integers). The individual elements of the set represent the actual time instants while the linear order represents the progression of time. The actual granularity of time is implementation-dependent[5]. Besides the data type for the time instants we also use all the other data types defined in standard SQL: strings, integers, floats, etc. As usual, these data types do not have an a-priori assigned interpretation. We summarily refer to those data types as *the uninterpreted constants*.

The relationships between the time instants and the uninterpreted constants are captured in a finite set of *temporal relations* stored in the database. Following the terminology of [9] we distinguish the *abstract* temporal databases from the *concrete* temporal databases:

Definition 3.1 (Abstract Temporal Database) A *signature* of a predicate symbol R is a tuple $(a_1 : t_1, \ldots, a_k : t_k)$ where a_i are distinct attribute names, t_i the corresponding attribute types, and k the arity of R. Attributes of type `time` are *temporal attributes*, the remaining attributes are *data attributes*. A *database schema* is a finite set of relational symbols $R_1, \ldots, R_k$ paired with their signatures. An *abstract temporal database* is a set of tables defined by a database schema.

In general we do not restrict the cardinality of the abstract temporal tables: we allow infinite tables as well. However, in order to define meaningful operations on the tables we require that the number of occurrences (duplicates) is finite for every distinct tuple.

Example 3.2 In the rest of the paper we use an abstract temporal database with the schema $\{$`indep`$(\text{Name}, \text{Year})\}$ as a running example. The particular instance of the `indep` relation we use in our examples captures independence of countries in Central Europe:

`indep`	
Name	Year
Poland	1025
...	...
Poland	1794
Poland	1918
...	...
Poland	1938
Poland	1945
...	...
...	
Czech Kingdom	1198
...	...
Czech Kingdom	1620
Czechoslovakia	1918
...	...
Czechoslovakia	1938
Czechoslovakia	1945
...	...
Czechoslovakia	1992
...	
Czech Republic	1995
...	...
...	...
Slovakia	1940
...	...
Slovakia	1944
Slovakia	1993
...	...
...	...

We do not impose any restrictions on the number of temporal attributes in relations (unlike, e.g., TSQL2 [17]). Indeed, in general we may want to record

[4] A dense linearly ordered time can be used with only a minor adjustment.

[5] For our purposes any fixed granularity will do.

relationships between different time instants as well as relationships between tuples of uninterpreted constants and a single time instant.

The abstract temporal databases provide a natural data model for modelling and querying temporal data. However, it would be impractical (and often impossible) to store the temporal databases as plain bags of their tuples: a particular tuple is often related to a large and possibly infinite set of time instants. Rather than storing all these tuples one by one, we use a *compact encoding* of sets of time instants. The choice of a particular encoding—in our case the *interval*-based encoding—defines the class of *concrete temporal databases*:

Definition 3.3 (Concrete Temporal Database) Let R be a relational symbol with signature E. A *concrete signature* corresponding to E is defined as a tuple of the attributes that contains (1) a for every data attribute $a \in E$ and (2) $t\mathtt{min}$ and $t\mathtt{max}$ for every temporal attribute $t \in E$. The attributes $t\mathtt{min}$ and $t\mathtt{max}$ denote endpoints of intervals. We denote the concrete signature of R by $\overline{E}$. A *concrete temporal database schema* corresponding to a given abstract database schema is a set of relation symbols and their concrete signatures derived from the signatures in the abstract database schema[6]. A *concrete temporal database* is a set of finite relations defined by a concrete database schema.

To capture the relationship between an abstract and a concrete temporal database we define a *semantic mapping* that maps a concrete temporal database to its meaning—an abstract temporal database. The meaning of a single concrete tuple $\mathbf{x} = (t\mathtt{min}, t\mathtt{max}, a_1, \ldots, a_k)$ is a bag of tuples $\|\mathbf{x}\| = \{(t, a_1, \ldots, a_k) : t\mathtt{min} \leq t \leq t\mathtt{max}\}$; analogously for tuples with multiple temporal attributes. The meaning $\|\mathbf{R}\|$ of a concrete relation $\mathbf{R}$ is the duplicate preserving union of $\|\mathbf{x}\|$ for all concrete $\mathbf{x} \in \mathbf{R}$. We say that $\mathbf{R}$ *encodes* $\|\mathbf{R}\|$. We extend the $\|.\|$ to concrete temporal databases in a natural way.

The *encodes* function also defines a subset of the abstract temporal databases that can be encoded using concrete temporal databases. We call this subset the *finitary temporal databases*. Note that the encoding is not unique and thus two distinct concrete temporal databases often encode the same abstract temporal database (cf. Example 2.1). We call such concrete temporal databases ($\|.\|$-)equivalent.

Example 3.4 The database instance from Example 3.2 is infinite. However, it is finitary: it can be encoded by the following concrete temporal database:

indep	
Name	Yearmin... Yearmax
Czech Kingdom	1198 ... 1620
Czechoslovakia	1918 ... 1938
Czechoslovakia	1945 ... 1992
Czech Republic	1993 ... ∞
...	
Slovakia	1940... 1944
Slovakia	1993... ∞
Poland	1025... 1794
Poland	1918... 1938
Poland	1945... ∞

All queries in the rest of the paper are evaluated over this database while preserving answers with respect to the original relation in Example 3.2.

4 The Language SQL/TP

In this section we define the syntax and semantics of SQL/TP. This includes the data definition, data query, and data manipulation parts of the language. In all

[6] We use the same names for both the abstract and concrete relations. The actual meaning of the symbol is always clear from the context.

three cases we show that SQL/TP is a natural syntactic extension of SQL over the abstract temporal databases. Moreover, the proposed semantics of SQL/TP is essentially identical to the semantics of SQL (safely) extended to potentially infinite tables.

4.1 Data Definition Language

We start with the *Data Definition Language*: it is essentially identical to standard SQL/92:

```
create table <rid> ( <signat> )
create view  <rid> ( <query> )
```

where `<rid>` is a table identifier and `<signat>` is a signature of the new table. For views the signature is derived from the signature of the `<query>` expression (cf. Section 4.2). The only *difference* is that the temporal attributes are declared using a new data type `time` that supports *modifiers* to determine *how the time instants are stored* in a concrete temporal table:

- `using points`: The time instants are stored as atomic values similarly to all other data types. This choice is suitable for representing single atomic events that happen at a specific time instant, e.g., when a particular tuple was inserted into the database.
- `using [bounded | unbounded] intervals`: Continuous sets of time instants associated with a particular data tuple are encoded using intervals. This encoding is suitable for representing durations of events, e.g., the valid time of a fact (which in reality is often an interval). The `bounded` and `unbounded` keywords specify if the $-\infty$ and ∞ may be used as endpoints of intervals. This choice affects, e.g., what aggregate operations are allowed for that particular attribute; cf. Section 4.2.

It is important to understand that these modifiers affect only the way the table is stored, not the semantics of the queries (similarly to specifying, e.g., a sort order or a key for the table).

In the future this list may grow to accommodate different encodings. The modifiers are the only place in SQL/TP where the syntax reflects the chosen encoding. The default modifier `unbounded time` is assumed for all temporal attributes unless explicitly stated otherwise.

Example 4.1 The table `indep` in Example 3.2 can be created as follows:

```
create table indep (name char(20),
                    year time using unbounded intervals)
```

In the rest of the paper we discuss only the interval-based encodings; encoding time instants by points does not introduce any problems over the traditional data types. In addition we assume the time instants can be represented by integers (using a fixed granularity) and we allow integer-like operations on the new data type so we do not get lost in superfluous syntax.

4.2 The Query Language

For sake of simplicity we discuss only a syntactic subset of full SQL/TP. This fact does not affect the generality of our proposal: it is an easy exercise to show that the proposed fragment forms a (first-order) complete query language [8]. Moreover, all representation independent SQL/Temporal queries, including queries with aggregation and universal subqueries, can be equivalently formulated in this fragment.

Syntax. The chosen syntactic subset of SQL/TP uses two basic syntactic constructs:

Select block. Similarly to the standard SQL the select block is the main building block of our query language. It has the usual form

```
select <slist> [from <flist>] [where <cond>] [group by <glist>]
```

where

- <slist> is a list of attribute identifiers, constants, and (aggregate) expressions with the possibility of renaming the output column using <sexp> as <id>[7],
- <flist> is a sequence of relation identifiers or subqueries, again with the usual possibility of assigning correlation names,
- <cond> is a selection condition built from atomic conditions using boolean connectives. The atomic conditions depend on the data types of the involved attributes: in the case of temporal attributes we allow conditions of the form <id> *op* <id> $+ C$ where $op \in \{<, \leq, =, \geq, >\}$, and C a constant denoting a *length* of a time period, and
- <glist> is a list of attribute identifiers that specifies how the result of the select block is grouped. The usual SQL rules that govern the grouping operations apply here as well.

We extend the definition of signature to SQL/TP expressions: The *signature* of an expression is tuple of names of attributes in the resulting table paired with their data types (including the modifiers).

Set Operations. Besides nesting queries in the `from` clause of the select block we can combine the individual select blocks using *set operations* as follows:

```
( <exp> ) <setop> ( <exp> )
```

where <setop> is one of the `union` (set union with duplicate elimination), `union all` (additive union), `except` (set difference with duplicate elimination), `except all` (monus), `intersect` (set intersection with duplicate elimination), and `intersect all` (duplicate preserving intersection). We require the signatures of both the expressions to match[8]. The resulting signature is the common signature of the expressions involved in the operation.

The proposed syntax omits two common SQL constructs: subqueries nested in the `where` clause and the `having` clause. Both these constructs can be expressed in the presented fragment using nesting in the `from` clause of the select block and can be considered to be a syntactic sugar.

---

[7] The columns defined using expressions or aggregation have to be given a name this way.

[8] SQL only requires the types to match. However, we require both the names of the attributes and their types to match. This is not a restriction as the renaming can be conveniently done within the `select` clauses.

To achieve signature compatibility for temporal attributes we allow the use of a special constant pseudo-relation `true(t: time)` true for all elements of the temporal domain. This relation allows us to pad the attribute lists involved in the set operations (cf. Section 4.3) and to express, e.g., the complementation over the temporal domain.

Semantics. SQL/TP is essentially SQL/92 extended with an additional data type `time`. The main *feature* of such an extension is that we can use the familiar *SQL-like semantics* over the class of the *abstract temporal databases*. This way we completely avoid all the problems connected with representation dependencies. Also, changes in the chosen encoding do not affect the syntax and semantics of queries.

However, we have to be careful when extending relational operations to infinite tables: we have to ensure that we never produce tables with infinite duplicates of a single tuple. It is easy to see that all the relational operations, with the exception of duplicate preserving projection, meet this requirement. However, the duplicate-preserving projection can produce such tables, e.g.:

$$\begin{aligned} \{(\text{“Poland”}, [1945, \infty])\} &\xrightarrow{\|\cdot\|} \{(\text{“Poland”}, n) : n \geq 1945\} \\ &\xrightarrow{\pi_1} \{(\text{“Poland”}), \ldots, (\text{“Poland”}), \ldots\} \end{aligned}$$

The result of the projection contains infinite duplication of the tuple (“Poland”). This cannot be allowed as other relational operators, e.g., the bag difference, are not well defined over such tables.

Closure over Interval-based Concrete Databases. While the above restriction guarantees a well defined semantics, it is too weak to guarantee closure of SQL/TP queries over the chosen class of concrete temporal databases. The main source of problems are the order dependencies among temporal attributes. Consider the following example:

Example 4.2 It is easy to find SQL/TP expressions that *do not* preserve closure over the class of finitary abstract temporal databases. Consider the expression:

```
Q: select  r1.name as name, r1.year as t1, r2.year as t2
   from    indep r1, indep r2
   where   r1.name = r2.name and r1.year < r2.year
```

The attributes `t1` and `t2` are *correlated* by an inequality `t1 < t2` *in the result of the query*:

$$\begin{array}{llll} \{(\text{“Poland”}, 1945, 1946), & (\text{“Poland”}, 1945, 1947), \ldots, & (\text{“Poland”}, 1945, 1950), \ldots & \\ & (\text{“Poland”}, 1946, 1947), \ldots, & (\text{“Poland”}, 1946, 1950), \ldots & \\ & & \ddots \qquad \vdots & \\ & & (\text{“Poland”}, 1949, 1950), \ldots\} & \end{array}$$

Obviously the triangle-like result can not be described by a product of intervals.

To avoid this problem we use the notion of *attribute independence*. Rather than a semantic definition of attribute independence [11] we use a syntactic inference system to detect attribute independence in a SQL/TP expression:

Definition 4.3 (Attribute Independence) Let t_1 and t_2 be two temporal attributes in the signature of a SQL/TP expression *exp*. We say that t_1 and t_2 are independent in *exp* if

1. *exp* is a base relation,
2. *exp* is a select block, t_1 and t_2 are names of t'_1 and t'_2 assigned in the select clause, t'_1 and t'_2 are independent in all expressions in the `from` list, and an order relationship between t'_1 and t'_2 is not implied by the `where` clause.
3. *exp* is (*e1*) setop (*e2*) and t_1 and t_2 are independent in both *e1* and *e2*.

In addition all data attributes (and point temporal attributes) are mutually independent.

For similar reasons we restrict the use of aggregate operations: we require the aggregated attribute to be independent of the `group by` attributes [10].

In addition we also restrict the use of duplicate-preserving projection on all temporal attributes encoded by intervals. We have already seen that duplicate-preserving projection is not possible for unbounded data types. On the other hand, for bounded data types we could implement the duplicate preserving projection by creating the appropriate number of copies of the remainder of a tuple. However, such an operation would make the query evaluation very inefficient and almost certainly unusable in practice. Consider the following example:

Example 4.4 Let $R(x,t) = \{(a, [0, 2^n - 1])\}$ be a concrete temporal relation where n is a large integer. Clearly the size of R (in bits) is $|a| + n$. However, the size of $\pi_x(R)$ is $2^n \cdot |a|$ as the result of duplicate preserving projection has to contain 2^n tuples (a).

Allowing such projections would cause an exponential blowup in the (space) complexity of query evaluation. Note that the duplicate preserving projection is used in SQL for two main reasons: (1) to avoid duplicate elimination or (2) to facilitate correct aggregates. The first use does not apply to SQL/TP—we deal with redundant duplicate elimination in the optimization phase of our compilation procedure. The aggregates are handled using a rewriting technique that allows us to avoid the duplicate-preserving projections[9]. This way we can evaluate a vast majority of representation-independent aggregate queries even under the above restriction: note that all other relational operations preserve duplicates (cf. Section 4.3). Therefore we exclude the duplicate-preserving projections of all temporal attributes encoded by intervals in order to maintain the polynomial complexity bound.

We define the SQL/TP queries to be the subset of SQL/TP expressions obeying the above rules. It is easy to verify that all SQL/TP queries preserve closure over the class of finitary temporal databases:

Theorem 4.5 *Let D be a finitary database and Q a SQL/TP query. Then $Q(D)$ is finitary.*

The requirement of attribute independence seems like a rather severe restriction. However, the independence is required only for the temporal attributes present in the signature of the top-level query, not for all temporal attributes that appear in the query. All the representation-independent TSQL2 queries, and all first order queries with a single temporal attribute in their signature in general, can be expressed as SQL/TP queries.

[9] This technique is out of the scope of this paper and is not needed for any of our examples.

Theorem 4.6 *The first-order fragment of SQL/TP is expressively equivalent to range restricted two-sorted first order logic (temporal relational calculus).*

We can also express queries shown not to be expressible in TRA [7], e.g., the query "is there a pair of distinct time instants, when exactly the same countries were independent?" [1, 20].

4.3 Examples of Queries

In this section we provide illustrative examples of SQL/TP queries. The examples are chosen to highlight the ease and naturality of formulating queries in SQL/TP. In addition some of the examples, e.g., example 3, can *not* be easily (and correctly) be formulated in TSQL2 or its derivatives.

1. The first example is a simple PSJ query "List all countries that were independent while Czech Kingdom was independent".

```
select  r1.name
from    indep r1, indep r2
where   r2.name = 'Czech Kingdom' and r1.year = r2.year
```

This query is a simple PSJ query (a single select block). Note also that the result is a standard non-temporal relation. The result of this query when evaluated over the database from Example 3.2 is:

```
name
--------------------
Czech Kingdom
Poland
```

2. Formulating more complicated queries in SQL/TP, e.g., the query "List all years when no country was independent", is easy and natural as well:

```
(select t as year from true) except (select year from indep)
```

The query is answered by complementation of the projection from the `indep` relation. Note the use of the `true` pseudo-relation to achieve signature compatibility. The result of the query is

```
  yearmin      yearmax
----------------------
-infinity        1024
     1795        1917
     1939        1939
```

While the output—a concrete table containing all the periods when no country was independent—has two columns, it is essential to understand that it is only a convenient and compact representation of an abstract table with a single column `Year`.

3. In addition to first-order queries, the aggregate operations in SQL/TP also naturally interact with the rest of the language, e.g., in the query "List all countries that became independent before Slovakia":

```
select  name
from    indep, ( select min(year) as y0
                 from   indep     where name = 'Slovakia' )
where   year < y0
```

The result is:

```
name
--------------------
Czech Kingdom
Czechoslovakia
Poland
```

4. SQL/TP also supports a natural way of aggregating over the temporal attributes: "For every country (that has been independent during the 20th century) list the number of years of independence within the 20th century"

```
select   name, count(year) as years
from     indep
where    1900  <=  year  <  2000
group by name
```

The aggregation is made possible by the `where` clause: it restricts the otherwise unbounded attribute `year`. The result is:

```
name                        years
------------------------------
Czechoslovakia                 67
Czech Republic                  7
Poland                         75
Slovakia                       11
```

Note that in query languages with interval-valued temporal attributes we would have to use a special syntactic construction to *measure* the size of the intervals explicitly.

5. Moreover, SQL/TP supports *grouping* by temporal attributes: "For every year list the number of independent countries (if any)":

```
select year, count(name) as numofc from indep group by year
```

The result is:

```
  yearmin      yearmax     numofc
---------------------------------
     1025         1197          1
     1198         1620          2
     1621         1794          1
     1918         1938          2
     1940         1944          1
     1945         1992          2
     1993     infinity          3
```

This query is quite hard to ask in temporal query languages that use coalescing implicitly: the input table is coalesced, and re-coalescing after the `name` column is eliminated leads to loosing the duplication we want to compute.

4.4 Database Updates

Besides considering the query language, in a truly practical approach we also need to address updates of temporal relations. We propose two constructs:

```
insert [all] into R ( <query> )
delete [all] from R ( <query> )
```

The updates have to preserve semantics with respect to the abstract temporal databases while manipulating only the concrete representation in a similar way queries do. The `delete` construction is more powerful than the SQL/92 version (as it handles duplication properly).

5 Evaluation of SQL/TP Statements

Starting with this section we focus on the second and third results of the paper: the compilation technique for point-based temporal queries to equivalent relational queries over interval-based concrete temporal databases. The subtle point here is that the resulting queries are *efficient*: they may refer only to the *active domain of the given concrete database*. This is not completely trivial as the semantics of the original queries is defined with respect to abstract temporal databases and a naive query evaluation procedure would indeed refer to all points in the active domain of the corresponding *abstract temporal database*—an immediate exponential blowup in the data complexity of the query evaluation.

While most approaches to query evaluation in temporal databases take the path of adding specialized temporal operations to a standard relational system, we take an alternative approach: we define a *translation procedure* that allows us to *compile* SQL/TP statements to standard SQL/92 statements. The translation utilizes the *quantifier elimination procedure* for linear order [21] to replace references to individual time instants in the queries with references to interval endpoints. In the rest of this section we give a sketch of the SQL/TP to SQL/92 translation. The translation is based on an extension of results in [19] to duplicate semantics and uses a nouveau normalization technique.

5.1 Data Definition Language

The translation of the data definition language statements is fairly simple: we merely convert the abstract signature to its concrete counterpart. The SQL/TP statement in Example 3.2 is translated to:

```
create table indep (name char(20), yearmin Time, yearmax Time)
```

where `Time` is a user defined type (UDT) for an integer like time. The data type `Time` is equivalent to `INTEGER`[10] extended with two special elements, $-\infty$ and ∞. We define the successor and order for this new type by lifting the operations from the `INTEGER` type.

5.2 The SQL/TP Queries

The crux of our approach lies in the translation of SQL/TP queries. The natural correctness criterion is the preservation of query semantics. This requirement is captured by the following diagram:

$$\begin{tikzcd}
\boxed{\text{Abstract TDB}} \arrow[d, "Q_{\text{SQL/TP}}"] & \boxed{\text{Concrete TDB}} \arrow[l, "\|\cdot\|"'] \arrow[d, "Q'_{\text{SQL}/92} = \text{compile}(Q)"] & \boxed{\text{Physical DB}} \arrow[l] \arrow[d, "\text{eval}(Q')"] \\
\boxed{\text{Abstract Relation}} & \boxed{\text{Concrete Relation}} \arrow[l, "\|\cdot\|"'] & \boxed{\text{Physical Relation}} \arrow[l]
\end{tikzcd}$$

[10] Often we can take advantage of a built-in data type provided by the RDBMS (e.g., `DATE` in DB/2).

We show that the proposed translation algorithm guarantees commutativity for the left part of the diagram. The commutativity of the right part is backed up by the reliability of the used relational system. The rest of this section gives a proof to the following theorem:

Theorem 5.1 *Let D be a concrete temporal database and Q a SQL/TP query. Then $Q(\|D\|) = \|\,\mathrm{compile}(Q)(D)\|$.*

Note that compile(Q) is executed at query compilation time—before the actual execution over the temporal database begins. Thus it does not affect the data complexity of the query evaluation algorithm. Before we present the steps of the translation itself, we introduce three auxiliary definitions.

The definition of SQL/TP queries requires the temporal attributes in the signature of a query to be independent. However, it does not prevent us from writing queries whose subqueries do not share this property. To deal with such attribute dependencies during the translation we introduce the notion of a *conditional query*:

Definition 5.2 (Conditional Query) Let Q be a SQL/92 query and φ a quantifier-free formula in the language of linear order such that t is a free variable of φ if and only if t`min` and t`max` are temporal attributes in the signature of Q. We call $Q\{\varphi\}$ a *conditional query*.

While the translation algorithm uses the conditional queries to translate subqueries of a SQL/TP query, the attribute independence of the top-level attributes guarantees that no such dependencies remain in the result of the translation.

The second challenge lies in the definition of relational operators that preserve semantics over the interval-based encoding. For this purpose we introduce a nouveau *normalization* technique. The idea behind the technique is quite simple:

Definition 5.3 Let $\{Q_1, \ldots, Q_k\}$ be a set of SQL/92 queries with compatible signatures such that X a subset of their data attributes and t a temporal attribute. $Q_1, \ldots, Q_k$ are *t-compatible on X* if for all concrete temporal databases D and all $0 < i \leq j \leq k$ whenever two concrete tuples $\mathbf{a} \in Q_i(D)$ and $\mathbf{b} \in Q_j(D)$ such that $\pi_X(\mathbf{a}) = \pi_X(\mathbf{b})$ then the sets $\pi_{\{t\}}(\|\mathbf{a}\|)$ and $\pi_{\{t\}}(\|\mathbf{b}\|)$ are identical or disjoint. $Q_1, \ldots, Q_k$ are *time-compatible* on X if $Q_1, \ldots, Q_k$ are t-compatible on X for all temporal attributes t in the common signature.

The definition of a time-compatible set of queries essentially says that if the data portion of a tuple is related to an interval in $Q_i(D)$ and to another interval in Q_j(D), then it is always the case that these two intervals coincide or are disjoint. This way we can guarantee the intervals behave like points with respect to set/bag operations (cf. Figures 2 and 3). This definition is non-trivial even for singleton sets of queries as the answers to queries are *bags* of tuples.

It is also easy to see that we can define a *normalization operation* that transforms a set arbitrary queries to a t-compatible set of ($\|.\|$-)equivalent queries. Moreover, this operation can be defined using a first-order query[11]:

Lemma 5.4 Let $\{Q_1, \ldots, Q_k\}$ be a set of SQL/92 queries with compatible signatures such that X a subset of their data attributes and t a temporal attribute. Then there are first-order queries $\mathsf{N}^t_X[Q_i; Q_1, \ldots, Q_k]$ such that

1. $\|Q_i(D)\| = \|\,\mathsf{N}^t_X[Q_i; Q_1, \ldots, Q_k](D)\|$ for all concrete databases D.
2. $\{\mathsf{N}^t_X[Q_i; Q_1, \ldots, Q_k] : 0 < i \leq k\}$ are t-compatible on X.

[11] Similarly to coalescing; a native implementation of the normalization can often be made more efficient.

Attribute	max(a)	min(a)	count(a)	sum(a)
data	max(a)	min(a)	sum(c)	sum$(a \cdot c)$
temporal	max$(a$max$)$	min$(a$min$)$	sum(c)	N/A

For every concrete tuple $\mathbf{x}$ in the table that the aggregation is applied to we let $c = \text{CNT}_G^\varphi(\mathbf{x})$. Note that the value of c is different for every tuple in the original table.

Fig. 1. Translation of Aggregate operations.

To define a time-compatible set of queries we use this lemma for all temporal attributes in the common signature. It is also easy to see that the normalization operation can be performed in $O(n \log n)$ where n is the combined size of the results of $Q_i(D)$.

The last obstacle is the translation of aggregate operations. To translate the aggregation operators correctly we need to know how many tuples are encoded by every single concrete tuple in the relation we aggregate over: we define a function CNT_G^φ for this purpose: it tells us how many duplicates would be in the $\|.\|$-image of the result of projecting a concrete tuple on G after applying the selection condition σ_φ. More formally:

Definition 5.5 Let E be an abstract signature, $G \subset E$, φ a quantifier free formula in the language of linear order over temporal variables in $E - G$, and $\mathbf{x}$ a concrete tuple in the signature $\overline{E}$. Then $\text{CNT}_G^\varphi(\mathbf{x}) = \text{card}(\sigma_\varphi \|\pi_{E-G}(\mathbf{x})\|)$.

Note that CNT_G^φ maps *concrete tuples* to natural numbers. However, if we used a dense model of time then CNT would be a measure on the sets of time instants and could return non-integral counts, e.g., 1.5 years. For details on aggregation and measures see [10].

Lemma 5.6 Given fixed E, $G \subset E$, and φ, the function CNT_G^φ can be defined using an integer expression over the value of $\mathbf{x}$.

The CNT function operates on single tuples and thus contributes only a constant to the overall data complexity of queries.

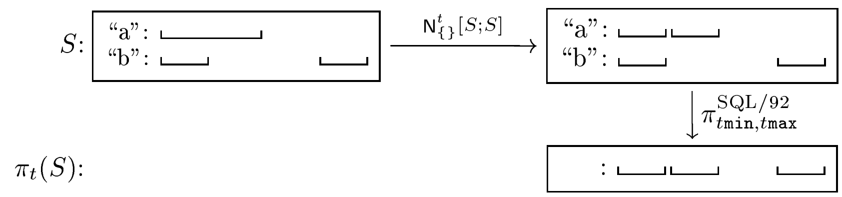
Now we are ready to proceed with the translation itself: every SQL/TP expression Q is translated to a set of conditional queries $Q_1\{\varphi_1\}, \ldots, Q_n\{\varphi_n\}$ while maintaining the invariant $Q(\|D\|) = \sigma_{\varphi_1}\|Q_1(D)\| \cup \ldots \cup \sigma_{\varphi_n}\|Q_n(D)\|$. The translation itself is defined inductively on the structure of the SQL/TP query.

Translation of the Select Block. Consider the SQL/TP statement:

$$\texttt{select [all]}\ X\ \texttt{from}\ E_1, \ldots, E_n\ \texttt{where}\ \varphi\ \texttt{group by}\ G$$

where X is the set of attributes in the answer, E_i subqueries or base table references, φ the selection condition, and G the set of grouping attributes. In addition, let A be the set of all aggregate expressions in X and E be the set of all attributes in $E_1, \ldots, E_n$. We assume that we have already translated the subqueries[12] to $Q_1\{\varphi_1\} \in \text{compile}(E_1), \ldots, Q_n\{\varphi_n\} \in \text{compile}(E_n)$. We compose these partial results to get a set of conditional queries equivalent to the original select block as follows (we proceed by translating every clause of the original select block one by one):

---
[12] for the base tables we merely add a trivial condition *true*.



S has data attribute x and a temporal attribute t. The boxes in the figure represent the t-x graphs of the involved tables. Similar technique is used for aggregation: it is easy to see that we could easily count duplicates on the normalized relation.

Fig. 2. Projection with Duplicate Elimination

`from` $E_1, \ldots, E_n$**:** For every $Q_1\{\varphi_1\}, \ldots, Q_n\{\varphi_n\}$ the `from` clause gives us a SQL/92 query

$$\mathtt{select}\ \overline{E}\ \mathtt{from}\ Q_1, \ldots, Q_n\{\psi\}$$

where $\psi = \varphi_1 \wedge \ldots \wedge \varphi_n$.

`where` φ**:** To apply the selection condition φ we need to determine the relationships between the endpoints of intervals, the corresponding point-valued attributes, and the selection formula. We use the quantifier elimination procedure for linear order to achieve this goal. Let $Q(E)\{\psi\}$ be the result of the previous step. We define

$$\psi_1 := \mathrm{QE}(\exists T(\psi \wedge \varphi \wedge \bigwedge_{t \in T} (t\mathtt{min} \leq t \leq t\mathtt{max})))$$
$$\psi_2 := \mathrm{QE}(\exists \overline{T}(\psi \wedge \varphi \wedge \bigwedge_{t \in T} (t\mathtt{min} \leq t \leq t\mathtt{max})))$$

where T is the set of all temporal attributes in E (encoded by intervals), $\overline{T}$ is the set of all attributes except those in T and constants, and QE is the quantifier elimination procedure for linear order. Now we define

$$\mathtt{select}\ \overline{E}\ \mathtt{from}\ Q\ \mathtt{where}\ \psi_1\{\psi_2\}$$

to be the result of applying the original `where` clause on $Q\{\psi\}$.

`group by` G**:** To apply grouping we first normalize the result of the previous step with respect to the attributes in G. Then we use the standard SQL/92 grouping construction (cf. Figure 2). Let $Q(E)\{\psi\}$ be the result from the previous step. As the attributes in G and $E - G$ are independent we can split ψ to ψ_1 involving only attributes in G and ψ_2 involving only attributes in $E - G$. We generate

$$\mathtt{select}\ \overline{G}, \overline{A}\ \mathtt{from}\ \mathsf{N}_G[Q;Q]\ \mathtt{group\ by}\ \overline{G}\{\psi_1\}$$

Note that the aggregates in A have to be transformed using Lemma 5.6 applied on E, G, and ψ_2.

`select` X**:** The translation of the final projection depends on the use of duplicate preserving vs. duplicate-eliminating projection. Let $Q(E)\{\psi\}$ be the result from the previous step.

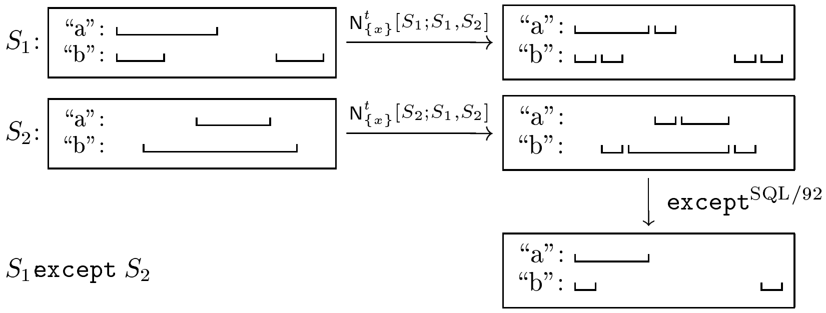


Fig. 3. Set Difference using Normalization.

- for `select` queries we get `select distinct` $\overline{X}$ `from` $\mathsf{N}_X[Q; Q]\{\psi\}$ and
- for `select all` queries we get `select` $\overline{X}$ `from` $Q\{\psi\}$

Note that only queries that use aggregation or duplicate elimination use the N operation.

For each of these steps we can easily verify that the transformation invariant is preserved. Moreover, in the actual implementation the subqueries generated by the above four steps are merged into as few nested blocks as possible, e.g., the first two steps can be always merged into a single select block, etc.

Translation of the Set Operations. The translation of the set operations follows similar pattern to the translation of the duplicate elimination: we need to find conditions under which the set operations on the encoding are equivalent to set operations on the abstract relations. Clearly a direct use of SQL/92 set operations does not preserve the semantics of SQL/TP queries.

Lemma 5.7 Let Q_1, Q_2 be two SQL/92 queries with a common signature and time-compatible on the set of all their data attributes. Then $\|Q_1\|\, op\, \|Q_2\| = \|Q_1\, op\, Q_2\|$ where *op* is one of the `union` [`all`], `except` [`all`], or `intersect` [`all`].

The above lemma is extended to conditional queries as follows: Let $Q_i\{\varphi_i\} \in \text{compile}(Q)$ for $i \in I$, $R_j\{\psi_j\} \in \text{compile}(R)$ for $j \in J$, and X the set of data attributes in the common signature. Then

$$\begin{array}{ll} Q\ \texttt{union}\ R & \mapsto \{Q_i\{\varphi_i \wedge \neg \bigvee_{j\in J} \psi_j\}, R_j\{\psi_j \wedge \neg \bigvee_{i\in I} \varphi_i\}, Q'_i\ \texttt{union}\ R'_j\{\varphi_i \wedge \psi_j\}\} \\ Q\ \texttt{except}\ R & \mapsto \{Q_i\{\varphi_i \wedge \neg \bigvee_{j\in J} \psi_j\}, Q'_i\ \texttt{except}\ R'_j\{\varphi_i \wedge \psi_j\}\} \\ Q\ \texttt{intersect}\ R & \mapsto \{Q'_i\ \texttt{intersect}\ R'_j\{\varphi_i \wedge \psi_j\}\} \end{array}$$

where $Q'_i = \mathsf{N}_X[Q_i; Q_i, R_j]$ and $R'_i = \mathsf{N}_X[R_j; Q_i, R_j]$. We can omit the normalization for the `union` operation. The duplicate-preserving operations are transformed analogously.

The result of the translation can unfortunately be exponential in the depth of nesting of the original query. Note that this does *not* affect the data complexity of the query evaluation as the translation is performed at compile time. Moreover, for large class of queries we can show:

Theorem 5.8 *Let Q be a query composed of attribute independent subqueries with size at most k for a fixed constant $k \in N$. Then* $|\text{compile}(Q)| \in O(|Q|)$.

Thus allowing *small* subqueries to violate the attribute independence requirement does not matter. Using this result we can show that, e.g., the first-order temporal logic queries can be efficiently translated to SQL/TP: all the temporal operators can be translated into small fixed-size subqueries and views with exactly one temporal attribute [3]. Similar result holds for all TRA [7] based languages.

6 Conclusion

We have shown that a high-level point-based approach to temporal extensions of SQL has many advantages over the common approaches that use interval-based attributes: simple syntax and semantics, meaningful aggregation, and possibilities of advanced query optimization. All this is achieved while maintaining efficient query evaluation over temporal databases based on interval encoding of timestamps. We have also shown that all representation independent TSQL2 queries are expressible in SQL/TP (follows from [19]).

Future Work. Our proposal is only a first step towards an implementation of SQL/TP on top of an ordinary RDBMS systems. There are still many open questions:

- Can we use more complex temporal domains? In our proposal we used a discrete linear order with a limited way of counting. Is it possible to use richer temporal domains while maintaining the properties of the proposed language? What are the tradeoffs?
- We have chosen an interval-based encoding of sets of time instants in the concrete data model. While this encoding can compactly describe periods of time, it fails, e.g., for periodic events. Is it possible to extend the encoding scheme and this way to enlarge the class of finitary temporal databases?
- What optimization techniques can be used in conjunction with our query translation procedure?
- How do we perform updates efficiently? The area of updates presents a completely new set of problems, the main problem being the in-place updates of the encoded temporal relations. This problem goes hand in hand with defining various normal forms [6, 19] of temporal relations and enforcing them over updates[13].
- Can the standard indices built-in relational systems aid the query evaluation based on the proposed compilation technique? What are the tradeoffs comparing to specialized indices?

Note that there are few answers to these questions even for the established temporal query languages like TSQL2 or SQL/Temporal. Note also that our technique allows us to reuse most of the efforts aimed towards boosting performance of temporal DBMS, e.g., the development of efficient temporal and spatial joins, and sophisticated access methods.

[13] In this paper we did not assume any particular normal form for the temporal relations.

References

1. Abiteboul, S., Herr, L., Van den Bussche, J. Temporal versus First-Order Logic to Query Temporal Databases. Proc. ACM PODS 1996, 49–57, 1996.
2. Allen, J. F. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
3. Bohlen, M. H., Chomicki, J., Snodgrass, R. T., Toman, D. Querying TSQL2 Databases with Temporal Logic. In Proc. EDBT'96, LNCS 1057, 325–341, 1996.
4. Böhlen, M. H., Jensen, C. S. Seamless Integration of Time into SQL. University of Aalborg, `http://www.cs.auc.dk/ boehlen/Software/Tiger/atsql.ps.gz`, 1996.
5. Böhlen, M. H., Jensen, C. S., Snodgrass, R. T. Evaluating and Enhancing the Completeness of TSQL2. Technical Report TR 95-5, Computer Science Department, University of Arizona, June 1995.
6. Böhlen, M. H., Snodgrass, R. T., Soo, M. D. Coalescing in Temporal Databases. Proc. *22nd Int. Conf. on Very Large Databases*, 180–191, 1996.
7. Clifford J., Croker A., Tuzhilin A. On Completeness of Historical Relational Query Languages. *ACM Transactions on Database Systems*, Vol. 19, No. 1, 64–116, 1994.
8. Codd, E. F. Relational completeness of database sublanguages. In Rustin, R.(ed.) *Courant Computer Science Symposium 6: Data Base Systems*, 65–98, Prentice-Hall, 1972.
9. Chomicki J. Temporal Query Languages: a Survey. Proc. *International Conference on Temporal Logic*, July 1994, Bonn, Germany, Springer-Verlag (LNAI 827), 506–534.
10. Chomicki, J., Kuper, G. M. Measuring Infinite Relations. Proc. ACM PODS 1995, 78–85, 1995.
11. Chomicki, J., Goldin, D. Q., Kuper, G. M. Variable Independence and Aggregation Closure. Proc. ACM PODS 1996, 40–48, 1996.
12. Date, C. J., Drawen, H. A Guide to the SQL Standard (3rd ed.), Addison–Welsley, 1993.
13. IBM Database 2, SQL Reference for common servers. IBM Corp., 1995.
14. Jensen, C. S., Snodgrass, R. T., Soo, M. J. Unification of Temporal Data Models. Proc. *9th Int. Conf. on Data Engineering*, 262–271, 1993.
15. Kanellakis, P. C., Kuper, G. M., Revesz, P.Z . Constraint Query Languages. Journal of Computer and System Sciences 51(1):26-52, 1995.
16. Snodgrass R. T. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.
17. Snodgrass R.T. (editor). *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 674+xxiv pages, 1995.
18. Snodgrass, R. T., Böhlen, M. H., Jensen C. S., Steiner, A. Adding Valid Time to SQL/Temporal, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2 21/11/96, (change proposal).
19. Toman, D. Point-based vs. Interval-based Temporal Query Languages Proc. ACM PODS 1996, 58–67, 1996.
20. Toman, D., Niwinski, D. First-Order Temporal Queries Inexpressible in Temporal Logic Proc. EDBT'96, Arpes, Bouzeghoub (eds.), LNCS 1057, 307–324, 1996.
21. Williams, H. P. Fourier–Motzkin Elimination Extension to Integer Programming Problems. In *Journal of Combinatorial Theory* (A) 21, 118-123, 1976.
22. Kabanza, F., Stevenne, J.-M., Wolper, P. Handling Infinite Temporal Data. JCSS 51(1): 3-17, 1995.

This article was processed using the LaTeX macro package with LLNCS style