Querying TSQL2 Databases with Temporal Logic

Michael H. Böhlen[1], Jan Chomicki[2],
Richard T. Snodgrass[3], and David Toman[4]

[1]Dept. of Mathematics and Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Ost, Denmark, boehlen@iesd.auc.dk

[2]Department of Computer Science, Monmouth University
West Long Branch, NJ 07764, chomicki@moncol.monmouth.edu

[3]Department of Computer Science, University of Arizona
Tucson, AZ 85721, rts@cs.arizona.edu

[4]Department of Computer Science, University of Toronto
Toronto, Ontario M5S 1A4, Canada

Abstract. We establish an exact correspondence between temporal logic and a subset of TSQL2, a consensus temporal extension of SQL–92. The translation from temporal logic to TSQL2 developed here enables a user to write high-level queries which can be evaluated against a space-efficient representation of the database. The reverse translation, also provided, makes it possible to characterize the expressive power of TSQL2. We demonstrate that temporal logic is equal in expressive power to a syntactically defined subset of TSQL2.

1 Introduction

In this paper, we bring together two research directions in temporal databases. The first direction is concerned with temporal extensions to calculus-based query languages such as SQL (e.g., [GN93, NA93, Sar93]). The issues addressed include space-efficient storage, effective implementation techniques, and handling large amounts of data. This approach includes the consensus temporal query language TSQL2 [Sno95], whose practical implementations should be forthcoming. The second direction is concerned with defining high-level query languages with logical semantics, e.g., temporal logic [TC90, GM91, CCT94]. The advantages of using logic languages come from their well-understood mathematical properties [GHR94]. Logic languages are easy to use and make algebraic query transformation possible [CT95]. For instance, temporal logic has been proposed as the language of choice for formulating temporal integrity constraints and triggers [Cho95, CT95, GL93, LS87, SW95].

The semantics of temporal logic queries is defined with respect to sequences of database states [GHR94]. In temporal databases we do not want to construct and store all the states explicitly. Instead, various proposals have associated with each fact a concise description of the set of points over which the fact holds, such as a period[1] [NA93, Sar90, Sno87, Tan86] or a finite union of periods

[1] We use the term 'period' in this paper rather than the term 'interval' commonly

[CC87, Gad88, Sno95]. We show here how to translate temporal logic queries into TSQL2, enabling the user to write high-level queries which will be evaluated against a space-efficient representation of the database. While translations of first order logic to SQL have been previously discussed [AHV95, VGT91], we know of no translations from temporal logic to a temporal query language.

We start with a discussion of the basic framework in Section 2. We define the syntax and semantics of the two languages in question, temporal logic and TSQL2. In Section 3 we give the mapping from temporal logic to TSQL2.[2] We conclude the section with an example and the discussion of some implementation issues. Section 4 discusses the reverse mapping, thereby relating the expressive power of (a subset of) TSQL2 and temporal logic.

2 Basic framework

Before comparing temporal logic and TSQL2 we have to set up a formal framework suitable to both languages. Time is considered to be *integer-like:* linear (totally ordered), discrete, bounded in the past, and infinite in the future. Our approach can be adopted to other kinds of time, e.g., dense, rational-like time, although some details of the mapping may in that case be different. We also take the *point-based* view which is predominant in the context of temporal logic. This view means that the truth-values of facts are associated with individual time points (also called instants). We assume a fixed time granularity.

We will consider only *valid-time*, which relates when facts are valid in reality [JCE+94]. In particular, *transaction time*, which relates when facts are stored in the database, is not considered.

2.1 Temporal logic

Temporal logic is an *abstract language*, i.e., a language which is defined with respect to abstract temporal databases [Cho94]. An *abstract temporal database*, in turn, is a database which captures the formal semantics of a temporal database without considering representation issues.

It is possible to view an abstract temporal database in several different but equivalent ways. We choose here the *timeslice* view (called *snapshot* in [Cho94]) in which every time instant is associated with a (finite) set of facts that hold at it. For integer-like time, this view leads to an infinite sequence of finite database states $(D_0, D_1, D_2, \ldots)$.

Example 1. Table 1 presents an example of an abstract temporal database, viewed as a sequence of states. The database represents information about Eastern European history, modeling the independence of various countries [Cho94]. Each fact indicates an independent nation and its capital. This relation will be used as a running example throughout the paper.

---

used in temporal logic because the latter term conflicts with SQL `INTERVAL`s, which are unanchored durations, such as 3 months.

[2] An implementation of the translation from temporal logic to TSQL2 is publicly available at `http://www.iesd.auc.dk/~boehlen/`.

Year	Timeslice
1025	$\{indep(`Poland`, `Gniezno`)\}$
...	...
1039	$\{indep(`Poland`, `Gniezno`)\}$
1040	$\{indep(`Poland`, `Cracow`)\}$
...	...
1197	$\{indep(`Poland`, `Cracow`)\}$
1198	$\{indep(`CzechKingdom`, `Prague`), indep(`Poland`, `Cracow`)\}$
...	...
1995	$\{indep(`CzechRepublic`, `Prague`), indep(`Poland`, `Warsaw`), indep(`Slovakia`, `Bratislava`)\}$
...	...

Table 1. Eastern European history: the abstract temporal database

Syntax. Temporal logic extends first order logic with binary temporal connectives **since** and **until**, and unary connectives ● ("previous" or "yesterday") and ○ ("next" or "tomorrow"). Informally, A **since** B is true in a state if A is true for states between when B was true and now (this state). A **until** B is true in a state if A will be true into the future until B will be true.

As usual, other temporal connectives can be defined in terms of these, e.g.,

$\blacklozenge A \equiv \texttt{true}\ \textbf{since}\ \texttt{A}$ (A was true sometime in the past)

$\Diamond A \equiv \texttt{true}\ \textbf{until}\ \texttt{A}$ (A will be true sometime in the future)

$\blacksquare A \equiv \neg\blacklozenge\neg A$ (A was true always in the past)

$\Box A \equiv \neg\Diamond\neg A$ (A will be true always in the future).

Example 2. Our first example is a query which does not relate different database states. The query

$$(\exists City)(indep(`Poland`, City) \wedge \neg(\exists City2)indep(`Slovakia`, City2))$$

determines all years when Poland but not Slovakia was an independent country, i.e., the times when the query evaluates to true.

Example 3. The second example relates different database states. The query

$$(indep(`Poland`, City) \wedge City \neq `Cracow`)\ \textbf{since}\ indep(`Poland`, `Cracow`)$$

returns the name of the city that superseded Cracow as Poland's capital and the years when this city was the capital.

Example 4. Consider the query [Cho94, p.515] "list all countries that lost and regained independence" over the abstract temporal database shown in Table 1. This is formulated in temporal logic as:

$$(\exists S1, S2)(\blacklozenge indep(X, S1) \wedge \Diamond indep(X, S2) \wedge (\forall S)\neg indep(X, S)).$$

For a country and a year to result, the country will have been independent in the past, will be independent in the future, but is currently not independent.

Semantics. An abstract temporal database is a sequence $D = (D_0, D_1, D_2, \ldots)$ of database states. Every database state D_i contains a relation (relation instance) r for each relation schema R. We define the semantics of temporal logic formulas in terms of a satisfaction relation $\models$ and a valuation ν (a valuation is a mapping from variables to constants):

- $D, \nu, i \models A$ iff A is atomic and $A/\nu \in D_i$ (where A/ν is the result of applying ν to the variables of A),
- $D, \nu, i \models \neg A$ iff $D, \nu, i \not\models A$,
- $D, \nu, i \models A \wedge B$ iff $D, \nu, i \models A$ and $D, \nu, i \models B$, similarly for $\vee$ and $\Rightarrow$,
- $D, \nu, i \models (\exists X)A$ iff for some c, $D, \nu[X \leftarrow c], i \models A$ where $\nu[X \leftarrow c]$ is a valuation identical to ν except that it maps X to c,
- $D, \nu, i \models (\forall X)A$ iff for all c, $D, \nu[X \leftarrow c], i \models A$,
- $D, \nu, i \models \bullet A$ iff $i > 0$ and $D, \nu, i-1 \models A$,
- $D, \nu, i \models \bigcirc A$ iff $D, \nu, i+1 \models A$,
- $D, \nu, i \models$ A **since** B iff $\exists j(j < i \wedge D, \nu, j \models B \wedge \forall k(j < k \leq i \rightarrow D, \nu, k \models A))$
- $D, \nu, i \models$ A **until** B iff $\exists j(j > i \wedge D, \nu, j \models B \wedge \forall k(i \leq k < j \rightarrow D, \nu, k \models A))$

The answer to a temporal logic query A in D is the set $\{(\nu, i) : D, \nu, i \models A\}$. Thus, temporal logic may be viewed as a natural extension of relational calculus.

As indicated by the example queries on the previous page, temporal logic provides a convenient means of expressing rather involved English queries in a natural way. However, the state-based semantics of temporal logic does not suggest an efficient implementation of such queries. A period-based implementation, in which the period over which each fact was valid is used directly in the evaluation, promises much faster execution.

2.2 TSQL2

TSQL2 [Sno95] is the consensus temporal extension of SQL–92 and, therefore, we use it as our target database query language when translating temporal logic. A *valid-time relation* is a relation where tuples are implicitly timestamped with periods[3].

Example 5. Table 2 contains a concrete TSQL2 relation representing the abstract temporal database shown in Table 1.

[3] In this paper, we use a slight variant of TSQL2 named *Applied TSQL2* (ATSQL2) [BJS95]. ATSQL2 modifies TSQL2 in a few minor ways. ATSQL2 timestamps tuples with periods rather than with temporal elements; ATSQL2 adds support for duplicates (though we will consider only ATSQL2 queries that remove duplicates); and ATSQL2 changes the syntax of the valid clause. We use ATSQL2 because the semantics of that language has been formally specified; only an informal specification of the semantics of TSQL2 has been given. Other than these changes, the languages are similar, and we will continue to refer to them under the rubric TSQL2.

indep

Country	Capital	VALID
Czech Kingdom	Prague	$[1198, 1620]$
Czechoslovakia	Prague	$[1918, 1938]$
Czechoslovakia	Prague	$[1945, 1992]$
Czech Republic	Prague	$[1993, \infty]$
Slovakia	Bratislava	$[1940, 1944]$
Slovakia	Bratislava	$[1993, \infty]$
Poland	Gniezno	$[1025, 1039]$
Poland	Cracow	$[1040, 1595]$
Poland	Warsaw	$[1596, 1794]$
Poland	Warsaw	$[1918, 1938]$
Poland	Warsaw	$[1945, \infty]$

Table 2. Eastern European history: the concrete TSQL2 relation

Syntax. TSQL2 extends the query language of SQL–92 [MS93] with the following constructs:

1. Syntactic constructs to manipulate timestamps (e.g., extract the start and end point of a period, construct a period out of two time points, etc.)[4].

2. Temporal built-in predicates, which can be used in the `WHERE` clause in order to specify temporal relationships between pairs of periods. To be consistent with SQL2, the relationships below have a somewhat different meaning than the identically-named relationships in [All83]. Notation: P^- for `BEGIN(`P`)` and P^+ for `END(`P`)`.

   - P_1 `=` P_2 iff $P_1^- = P_2^-$ and $P_1^+ = P_2^+$
   - P_1 `CONTAINS` P_2 iff $P_1^- \leq P_2^-$ and $P_1^+ \geq P_2^+$
   - P_1 `MEETS` P_2 iff $succ(P_1^+) = P_2^-$
   - P_1 `OVERLAPS` P_2 iff $P_1^- \leq P_2^+$ and $P_2^- \leq P_1^+$
   - P_1 `PRECEDES` P_2 iff $P_1^+ < P_2^-$

   Also, period endpoints can be compared directly using `PRECEDES`. In this way, all the relationships in [All83] can be expressed.

3. A valid clause, which can be placed in front of queries.[5] It is to specify whether a query expression should be evaluated with temporal semantics (no valid clause) or with standard Codd semantics (valid clause). Intuitively, temporal semantics corresponds to snapshot reducibility [Sno87] which means that, conceptually, the respective query is evaluated over every

[4] These features are briefly discussed where used the first time.

[5] The same syntactic extension is allowed for queries in the `FROM` clause defining a derived table. With auxiliary tables or views such queries can be rewritten so that the valid clause only occurs at "the outermost level".

snapshot of a temporal database. The valid clause comes in two different flavors. If it is of the form `VALID SNAPSHOT`, a snapshot relation is returned. Otherwise, it is of the form `VALID` *expr* in which case a valid-time relation is returned with valid-time defined by *expr*.

4. `(PERIOD)` may follow a query expression or a relation name in a from clause, specifying that the result be *coalesced*, that is, tuples with identical explicit attribute values whose valid-times overlap or are adjacent are merged into a single tuple, with a period equal to the union of the periods of the original tuples. As a side-effect, duplicates are eliminated. We use `(PERIOD)` throughout because temporal logic does not allow duplicates or uncoalesced periods.

5. Other facilities not relevant here, including temporal indeterminacy, schema evolution, user-defined granularities, and extensible literal syntax.

Semantics. TSQL2 has been given a formal denotational semantics that maps TSQL2 statements to (temporal) relational algebra expressions [BJS95].

Example 6. In order to determine the name of the city that superseded Cracow as Poland's capital (c.f., Example 3), different database states have to be related. In TSQL2 this means that we have to specify a valid clause (in order to override snapshot reducibility) and we also have to specify the required temporal relationship. This results in the following TSQL2 query Q_1:

```
VALID VALID(i1)
  SELECT i1.Capital
  FROM indep(PERIOD) AS i1, indep(PERIOD) AS i2
  WHERE i1.Country = 'Poland'
   AND i2.Country = 'Poland' AND i2.Capital = 'Cracow'
   AND VALID(i2) MEETS VALID(i1)
```

Example 7. The formulation of a query becomes even simpler if it can be answered by looking at single snapshots. In this case the user can simply ignore time when formulating a query, as illustrated in the following query Q_2, which determines all period(s) when Poland but not Slovakia was independent (c.f., Example 2):

```
(SELECT i1.Country
  FROM indep(PERIOD) AS i1
  WHERE i1.Country = 'Poland'
   AND NOT EXISTS (
     SELECT *
     FROM indep(PERIOD) AS i2
     WHERE i2.Country = 'Slovakia'))(PERIOD)
```

3 Mapping Temporal Logic to TSQL2

A mapping from temporal logic to TSQL2 is useful for two reasons. First it relates the two languages and, thus, their expressive power. Second it yields an efficient implementation for temporal logic formulas using a translation to

TSQL2 that efficiently encodes identical adjacent facts. Also TSQL2 queries can be optimized.

Before we can describe the actual mapping of temporal formulas to TSQL2 we need to establish a relationship between the databases over which our translation is well defined. Not every abstract temporal database can be represented as a TSQL2 database. For example, the database that has a single fact $p(a)$ in every even-numbered state and whose every odd-numbered state is empty cannot be represented in TSQL2.

Definition 1. Let $D = (D_1, D_2, \ldots)$ be an abstract temporal database. The *support* of a temporal logic formula A under a valuation ν is the set

$$\{i : D, \nu, i \models A\}.$$

The support for ground formulas (e.g., facts) does not depend on the valuation. The definition of the support allows us to define the class of abstract temporal databases we are interested in.

Definition 2. An abstract temporal database is *finitary* if it contains a finite number of facts and the support of every fact can be represented as a finite union of periods.

Proposition 3. *Every TSQL2 database represents a finitary abstract temporal database and every finitary abstract temporal database can be represented by a TSQL2 database.*

The previous observations are used to define the translation of temporal logic formulas to TSQL2 and prove its correctness. The translation uses an extension of existing methods for translating first-order logic formulas to SQL–92, e.g., [VGT91, AHV95, Wüt91] as one of its steps. We also give a syntactic criterion for identifying (a subset of) domain independent formulas of temporal logic.

3.1 Temporal Logic to TSQL2 Translation

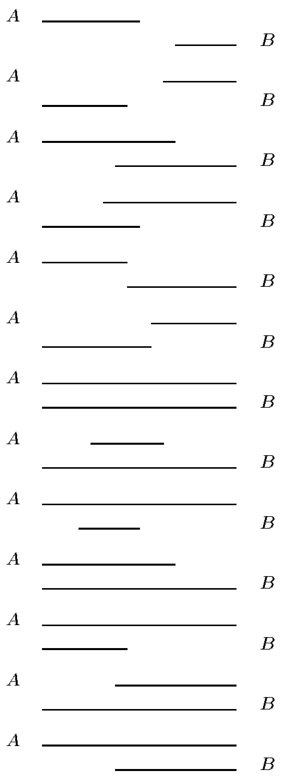
The translation of temporal logic formulas to TSQL2 is defined by induction on the structure of the formula. Temporal logic formulas can be thought of as first-order formulas augmented by additional *temporal* connectives **since**, **until**, $\bullet$, and $\circ$. This observation allows us to define the translation process in two steps:

1. mapping of temporal connectives, and
2. mapping of maximal sub-formulas not containing temporal connectives.

The whole translation procedure then works inductively on the structure of the given query: It first computes the TSQL2 equivalents of the maximal first-order subformulas of the query. The results are then used in the definitions of the translations of temporal connectives to TSQL2 views. These views then *replace* the original temporal subformula (by a virtual relation name). The process is repeated until the whole formula is translated.

We first describe the mapping of the temporal connectives to TSQL2. This mapping links the translations of the (essentially) first-order pieces of the original query together.

Mapping since and until. Figure 1 graphically illustrates the semantics of **since** and **until**. We have listed all possible temporal relationships [All83] between the truth periods of two formulas A and B. For each relationship we have determined the truth period of A **since** B and A **until** B respectively. (A^-/A^+ denotes the start/end point of the truth period of A.) More formally the truth periods of A **since** B and A **until** B are defined as follows.



Temporal relationship between formulas A and B	Temporal logic formula F	Truth period of formula F
1	A **since** B	[ ]
	A **until** B	[ ]
2	A **since** B	[ ]
	A **until** B	[ ]
3	A **since** B	$[succ(B^-), A^+]$
	A **until** B	$[A^-, A^+]$
4	A **since** B	$[A^-, A^+]$
	A **until** B	$[A^-, pred(B^+)]$
5	A **since** B	[ ]
	A **until** B	$[A^-, A^+]$
6	A **since** B	$[A^-, A^+]$
	A **until** B	[ ]
7	A **since** B	$[succ(B^-), A^+]$
	A **until** B	$[A^-, pred(B^+)]$
8	A **since** B	$[A^-, A^+]$
	A **until** B	$[A^-, A^+]$
9	A **since** B	$[succ(B^-), A^+]$
	A **until** B	$[A^-, pred(B^+)]$
10	A **since** B	$[succ(B^-), A^+]$
	A **until** B	$[A^-, A^+]$
11	A **since** B	$[succ(B^-), A^+]$
	A **until** B	$[A^-, pred(B^+)]$
12	A **since** B	$[A^-, A^+]$
	A **until** B	$[A^-, pred(B^+)]$
13	A **since** B	$[succ(B^-), A^+]$
	A **until** B	$[A^-, pred(B^+)]$

Fig. 1. Semantics of **since** and **until**

$$\text{A } \textbf{since} \text{ B} \mapsto [max(A^-, succ(B^-)), A^+] \quad \text{for } max(A^-, succ(B^-)) \leq A^+ \text{ and } B^+ \geq A^-$$

$$\text{A } \textbf{until} \text{ B} \mapsto [A^-, min(A^+, pred(B^+))] \quad \text{for } A^- \leq min(A^+, pred(B^+)) \text{ and } A^+ \geq B^-$$

The reader may verify that these general expressions evaluated on any particular relationship given in Figure 1 result in the correct truth period.

These expressions can be translated to TSQL2 straightforwardly. The valid clause is used to specify the final timestamp (and to prevent snapshot reducibility), whereas the conditions are translated into appropriate `WHERE` clause conditions. More precisely, A **since** B is translated to[6]

```
VALID PERIOD(LAST(BEGIN(VALID(a0)),BEGIN(VALID(a1))+1), END(VALID(a0)))
  SELECT ...
  FROM A'(PERIOD) AS a0, B'(PERIOD) AS a1
  WHERE LAST(BEGIN(VALID(a0)),BEGIN(VALID(a1))+1) <= END(VALID(a0))
   AND BEGIN(VALID(a0)) <= END(VALID(a1))
   AND ...
```

whereas A **until** B is translated to

```
VALID PERIOD(BEGIN(VALID(a0)), FIRST(END(VALID(a0)),END(VALID(a1)))-1)
  SELECT ...
  FROM A'(PERIOD) AS a0, B'(PERIOD) AS a1
  WHERE BEGIN(VALID(a0)) <= FIRST(END(VALID(a0)),END(VALID(a1))-1)
   AND BEGIN(VALID(a1)) <= END(VALID(a0))
   AND ...
```

The `SELECT` list of the TSQL2 statements is determined from the free variables occurring in either A or B. Variables used in A and B give rise to further `WHERE` clause conditions. We get A' by applying the translation recursively to A and B' by applying the translation recursively to B.

Mapping $\bullet$ and $\circ$. The mapping of the connectives $\bullet A$ and $\circ A$ is defined as follows: First we define the truth periods for $\bullet A$ and $\circ A$ with respect to the truth period of A:

$$\bullet A \mapsto [succ(A^-), succ(A^+)]$$
$$\circ A \mapsto [pred(A^-), pred(A^+)]$$

The result is translated to TSQL2 using a definition of the corresponding valid-time clause that shifts the valid-time period by one in the appropriate direction. The translation for $\bullet A$ looks like

```
VALID VALID(a0)+1
  SELECT ...
  FROM A'(PERIOD) AS a0
```

and the translation for $\circ A$ is

```
VALID VALID(a0)-1
  SELECT ...
  FROM A'(PERIOD) AS a0
```

The `SELECT` list is again obtained from the set of free variables in A, and A' is the TSQL2 translation of A.

---

6 `PERIOD`(x,y) takes two timepoints x and y, and returns a period. `BEGIN/END` returns the start/end point of a period. `FIRST` and `LAST` return the minimum and maximum timepoint out of a pair of timepoints, respectively. Finally, we assume here that the valid-time is at a granularity of a year. Thus `+1` is shorthand for `+INTERVAL '1' YEAR` and `-1` for `-INTERVAL '1' YEAR`.

Mapping of first-order (sub-)formulas. The mapping of first-order formulas to relational algebra has been described in several papers and books, e.g., [VGT91, AHV95]. As our target language is (T)SQL2 rather than relational algebra, we map maximal first-order subformulas to directly to SQL [Wüt91], thereby exploiting the syntactic features of the latter and achieving efficient SQL queries.

3.2 Domain independence

Similarly to the first-order case [VGT91, AHV95], not all formulas expressible in temporal logic are domain-independent. We identify (a subset of) the domain-independent formulas of temporal logic using an extension of the syntactic criteria defined for first-order formulas.

Proposition 4. *Let φ be a temporal formula and* ALWD *be a domain-independence criterion for first-order formulas. If*

1. ALWD(FOL(φ)),
2. ALWD(FOL(A)) *and* ALWD(FOL(B)) *for every subformula of φ that has the form A* **until** *B or A* **since** *B, and*
3. ALWD(FOL(A)) *for every subformula of φ of the form op A where op is one of* $\{\blacklozenge, \lozenge, \blacksquare, \square, \bullet, \circ\}$

where FOL *is a mapping that replaces all occurrences of temporal subformulas by (imaginary) database relations with the same sets of free variables, then φ is domain-independent.*

The domain-independence needs to be extended to the temporal domain as well. We need to show that every tuple in the result of our query is associated with a finite union of periods. However, it is easy to show that:

Theorem 5. *For any finitary temporal database and a fixed valuation the support of every temporal logic formula can be represented by a finite union of periods.*

Thus the application of boolean operators, temporal operators, and quantifiers preserves the *finitary* property of relations. This result shows that all the intermediate results can be represented by finite unions of periods (and thus evaluated properly using TSQL2).

3.3 Correctness of the translation

Proposition 4 guarantees that at every point of the transformation process we only have to deal with domain-independent formulas (i.e., all first order variables are range-restricted). Thus there are only finitely many valuations (for any finitary temporal database) at that point. Thus

Theorem 6. *For every temporal logic formula φ satisfying the assumptions of Proposition 4 and for every finitary abstract temporal database D the following* *diagram commutes:*

$$\begin{array}{ccc} D & \xrightarrow{\varphi} & R \\ \downarrow & & \downarrow \\ D^{TSQL2} & \xrightarrow{\varphi^{TSQL2}} & R^{TSQL2} \end{array}$$

where D^{TSQL2} is the TSQL2 equivalent of D, φ^{TSQL2} is the translation of the temporal logic query φ, and R^{TSQL2} is the TSQL2 variant of the result of the query.

3.4 Deriving specialized mappings

Based on the translation of **since** and **until**, the mapping of other temporal connectives can be defined. While theoretically feasible such an approach may be cumbersome in practice as it leads to unnecessarily complicated TSQL2 statements.

Mapping $\blacklozenge$. We illustrate how the definition of **since** can be used to derive an efficient special purpose mapping for $\blacklozenge$. The formula $\blacklozenge B$ is equivalent to `true` **since** `B`. Therefore we take the definition of A **since** B (Section 3.1) and substitute A by `true`. We notice that the truth period of `true` is the whole time line which means that `BEGIN(VALID(a0))` evaluates to 0 (beginning of time) and `END(VALID(a0))` evaluates to ∞ (end of time). After the obvious simplifications we obtain:

```
VALID PERIOD(BEGIN(VALID(a1))+1, TIMESTAMP 'forever')
  SELECT ...
  FROM B'(PERIOD) AS a1
```

which is considerably less complex than the original statement. Similarly, we can use the definition of **until** to derive a mapping for $\Diamond$.

Mapping $\blacksquare$. For $\blacksquare A$, one can rewrite it as $\neg\blacklozenge\neg A$ and use the approach presented above. Unfortunately, this approach is not very practical as it may lead to formulas that cannot be translated (e.g., $\neg\blacklozenge\neg p(X)$ versus $\blacksquare p(X)$). Therefore we derive a TSQL2 translation for $\blacksquare A$ from the definition

$$D, \nu, i \models \blacksquare A \text{ iff } \forall j(j < i \rightarrow D, \nu, j \models A)$$

Assuming bounded time in the past, this can be easily expressed in TSQL2:

```
VALID PERIOD(BEGIN(VALID(a0)), END(VALID(a0))+1)
  SELECT ...
  FROM A'(PERIOD) AS a0
  WHERE BEGIN(VALID(a0)) = TIMESTAMP 'beginning'
```

By analogy, a special purpose mapping for $\Box A$ can be derived.

3.5 Example

Consider the query "list all countries that lost and regained independence" (Example 4) formulated in temporal logic as:

$$(\exists S1, S2)(\blacklozenge indep(X, S1) \wedge \Diamond indep(X, S2) \wedge (\forall S)(\neg indep(X, S))).$$

To simplify the illustration of the translation we break up the formula into a set of auxiliary rules (views):

$aux\_view1(X, S1) \leftarrow \blacklozenge indep(X, S1).$
$aux\_view2(X, S2) \leftarrow \Diamond indep(X, S2).$

$(\exists S1, S2) aux\_view1(X, S1) \wedge aux\_view2(X, S2) \wedge (\forall S)(\neg indep(X, S)).$

We translate the first rule to

```
VALID PERIOD(BEGIN(VALID(a0))+1, TIMESTAMP 'forever')
  SELECT a0.Country, a0.Capital
  FROM indep(PERIOD) AS a0
```

and the second rule to

```
VALID PERIOD(TIMESTAMP 'beginning', END(VALID(a1))-1)
  SELECT a1.Country, a1.Capital
  FROM indep(PERIOD) AS a1
```

The main query is then translated to

```
SELECT a2.Country AS Country
FROM aux_view1(PERIOD) AS a2, aux_view2(PERIOD) AS a3
WHERE a2.Country = a3.Country
AND NOT EXISTS (
  SELECT *
  FROM indep(PERIOD) AS a4
  WHERE a4.Country = a2.Country)
```

Note that this last step is identical to the translation from first order logic to SQL. Because temporal logic and TSQL2 handle the temporal dimension of snapshot-reducible queries automatically, the translation of temporal logic formulas that do not contain temporal connectives degenerates to the translation of first order logic to SQL.

4 Mapping TSQL2 to temporal logic

Establishing a mapping between TSQL2 and temporal logic is less important from a practical point of view than establishing the mapping in the other direction, as described in the previous section. However, the former mapping makes it possible to study the expressive power of TSQL2 as a query language.

Definition 7. A TSQL2 query is *pure* if:

1. It does not use aggregate operators.
2. Coalescing of periods is forced using `(PERIOD)`. As a side-effect, this ensures that no duplicates are generated.

The idea is to use only those features of SQL that can be mapped to relational calculus or algebra.

Definition 8. A TSQL2 query is *local* if:

1. In every subclause of a `SELECT`, all the references of the form `VALID(`v`)` refer to a tuple variable v of the `FROM` clause of this particular `SELECT`. (There is

no similar requirement for nontemporal attributes.) This implies that nested `SELECT` clauses cannot refer to the valid-times of correlation names specified in the `FROM` clause of an enclosing `SELECT`.

2. The only arithmetic expressions in which `VALID`(v) can appear are of the form `VALID`$(v) \pm k$ for an integer k.
3. No `VALID SNAPSHOT` clauses appear.

Example 8. The following TSQL2 query is nonlocal.

```
(VALID VALID(a)
  SELECT * FROM a AS a
  WHERE NOT EXISTS
    (SELECT * FROM b AS b
     WHERE VALID(a) MEETS VALID(b) AND a.X=b.Z))(PERIOD)
```

Our mapping maps pure local TSQL2 queries to temporal logic formulas. Its main idea is illustrated by the following example.

Example 9. Consider the following (pure local) TSQL2 query.

```
(VALID PERIOD(BEGIN(VALID(b)),END(VALID(c)))
  SELECT *
  FROM a AS a, b AS b, c AS c
  WHERE VALID(a) OVERLAPS VALID(b)
    AND VALID(c) OVERLAPS VALID(b)
    AND a.X=b.Z)(PERIOD)
```

Assume that `a` has two attributes: `X` and `Y`, `b` one attribute `Z`, and `c` also one attribute `W`.

We extend previous notation to apply to tuple variables as follows: $\mathtt{x}^-$ denotes `BEGIN(VALID(x))` and $\mathtt{x}^+$ denotes `END(VALID(x))`. Based on the `WHERE` clause, period endpoints have to be partially ordered in the following way:

$$\mathtt{a}^- \le \mathtt{b}^+ \wedge \mathtt{b}^- \le \mathtt{a}^+ \wedge \mathtt{c}^- \le \mathtt{b}^+ \wedge \mathtt{b}^- \le \mathtt{c}^+.$$

Now consider all linear orders of endpoints that are consistent with the above partial order, for example, the linear order O_1:

$$\mathtt{a}^- < \mathtt{c}^- < \mathtt{b}^- < \mathtt{c}^+ < \mathtt{a}^+ < \mathtt{b}^+.$$

Given a linear order O, every period with endpoints that are successive elements in O is called *nondecomposable*. Notice that in each such period and for each fixed valuation the truth values of $a(X,Y)$, $b(Z)$ and $c(W)$ do not change. For each such period P in a given linear order O, denote by α_P^O the conjunction of $a(X,Y)$, $b(Z)$, and $c(W)$ or their negations that is true over all the points in P. The formula α_P^O will be called the *local characteristic* of the period P in O. For example, $\alpha_{[\mathtt{b}^-,\mathtt{c}^+]}^{O_1}$ is $a(X,Y) \wedge b(Z) \wedge C(W)$. We also define the *global characteristic* of P in a given linear order O as the temporal logic formula β_P^O that encodes the given linear order of endpoints and is true exactly over P. The order O_1 leads to the formula $\beta_{[\mathtt{b}^-,\mathtt{c}^+]}^{O_1}$ which is a conjunction of

$$(a(X,Y) \wedge b(Z) \wedge c(W)) \textbf{ until}$$
$$((a(X,Y) \wedge b(Z) \wedge \neg c(W)) \textbf{ until } (\neg a(X,Y) \wedge b(Z) \wedge \neg c(W)))$$

and

$$(a(X,Y) \wedge b(Z) \wedge c(W)) \textbf{ since}$$
$$((a(X,Y) \wedge \neg b(Z) \wedge c(W)) \textbf{ since } (a(X,Y) \wedge \neg b(Z) \wedge \neg c(W)))$$

The temporal logic formula corresponding to the query with `VALID` period P[7] is obtained as the conjunction of the nontemporal condition in the `WHERE` clause (here: $X = Z$) and the disjunction of all the formulas formulas β_P^O where O is a linear order consistent with the partial order given by the `WHERE` clause.

Theorem 9. *For every pure local TSQL2 query Q, there is a temporal logic formula ϕ_Q such that for every TSQL2 database D, a tuple $\bar{a}$ timestamped by an period i belongs to the answer of Q over D iff $D', \nu, t \models \phi_Q$ for every timepoint t in i (where D' is the abstract temporal database corresponding to D and ν is the valuation that maps the free variables of ϕ_Q to $\bar{a}$).*

Proof. (sketch) The formula ϕ_Q is defined inductively. For a base relation p with n attributes, ϕ_Q is just $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are different variables. For a `VALID` P `... SELECT` where P is a nondecomposable period, ϕ_Q is obtained as a disjunction of all the global characteristics β_P^O where O is a linear order consistent with the partial order given by the `WHERE` clause, as in Example 9 (all of TSQL2 built-in temporal predicates can be handled in this way).

There are several additional points that need to be considered. First, there may be more than one possible partial order of endpoints obtained from the `WHERE` clause. The resulting formula is obtained then as a disjunction of formulas corresponding to individual partial orders. Second, the period in the `VALID` clause may be decomposable. Then the TSQL2 query may be viewed as a finite union of TSQL2 queries in which such periods are nondecomposable. Third, temporal expressions on the valid-times have to be handled in a special way. In every linear order, one needs to consider not only period endpoints but also the appropriate neighboring points (predecessors and successors). As a result, local characteristics may now contain also $\bullet$ and $\circ$. For instance, in Example 9 the local characteristic of the period $[succ(\mathtt{b}^-), \mathtt{c}^+]$ should be $a(X,Y) \wedge b(Z) \wedge c(W) \wedge \bullet b(Z)$ and the global characteristic should be changed similarly. Fourth, temporal constants, e.g., 2 can be encoded using $\bullet$. Namely, we define inductively the formula n_i which is true exactly in the state D_i:

$$n_0 \stackrel{\text{def}}{\equiv} \neg \bullet \mathtt{true}$$
$$n_{i+1} \stackrel{\text{def}}{\equiv} \bullet n_i.$$

To deal with unanchored spans, e.g., "3 instants", we introduce sufficiently many (3 in the example) additional points associated with the formula `true` into the partial ordering and construct the local characteristic appropriately. Finally, anchored spans are dealt with using the combination of the above techniques. In all cases, one produces the characteristics in essentially the same way: by encoding the linear order in temporal logic.

Moreover, nontemporal conditions in the `WHERE` clause, `NOT`, and `EXISTS` are translated as in the standard translation from SQL to (domain) relational calculus. For the attributes not in the `SELECT` list, appropriate existential quantifiers

[7] If no `VALID` clause is present the intersection of all valid periods corresponding to the `FROM` list is assumed.

are added to the formula. Finally, `SELECT` without a `VALID` clause is translated using the standard translation from SQL to domain relational calculus.

The translation from temporal logic to TSQL2 presented in the previous section produces pure local TSQL2 queries. Thus:

Corollary 10. *Temporal logic and pure local TSQL2 have the same expressive power as query languages.*

There is a subtle point here: the above translation produces temporal logic formulas that are domain-independent. However, not every such formula satisfies the assumptions of Proposition 4 and is thus amenable to the translation back to TSQL2. We conjecture that this gap may be closed by providing a more sophisticated translation from temporal logic to TSQL2.

The following is a natural next question to ask: Is there a logical query language equivalent to full TSQL2? The lack of aggregates in temporal logic can be remedied by a syntactic extension of the language, along the lines of one proposed for relational calculus [Klu82]. The requirement of maximal periods is more fundamental. In fact, allowing noncoalesced periods calls for a temporal logic that is not point- but period-based [Tom95]. Thus in this case, there can be no translation from full TSQL2 to the temporal logic discussed in this paper, even for local queries.

The restriction to local queries is also critical. Pure TSQL2 has the same expressive power as two-sorted first-order logic in which there is a separate sort for time [Tom95]. It has been recently shown [AHVdB95, TN96] that temporal logic is strictly less expressive than the above two-sorted logic. Thus, there can be no translation from TSQL2 to temporal logic that works for all pure queries.

5 Summary

We have established an exact correspondence between temporal logic and a syntactically defined subset of TSQL2. The translation from temporal logic to TSQL2 allows the efficient implementation of temporal logic queries within a temporal database management system supporting TSQL2.

Future work includes extending the class of allowed temporal logic formulas (which will also require extensions to the translation to TSQL2), and extending temporal logic and the translation to support aggregates. Also interesting would be an adaptation of our approach to a dense domain. This would require first extending TSQL2 to such a domain, including support for half-open and open periods, and then extending the mapping introduced here. Finally, a translation from two-sorted first-order logic to TSQL2, which is of clear practical interest, seems considerably more complicated than the translation from temporal logic to TSQL2 given in the present paper.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[AHVdB95] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal Connectives versus Explicit Timestamps in Temporal Query Languages (unpublished manuscript).

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 16(11):832–843, 1983.

[BJS95] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating and Enhancing the Completeness of TSQL2. Technical Report TR 95-5, Computer Science Department, University of Arizona, June 1995.

[CC87] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra based on Lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537, Los Angeles, CA, February 1987.

[CCT94] J. Clifford, A. Croker, and A. Tuzhilin. On Completeness of Historical Relational Query Languages. *ACM Transactions on Database Systems*, 19(1):64–116, March 1994.

[Cho94] J. Chomicki. Temporal Query Languages: a Survey. *Proceedings of the First International Conference on Temporal Logic*, pages 506–534, 1994.

[Cho95] J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems*, (20) 2, 149–186, 1995.

[CT95] J. Chomicki and D. Toman. Implementing Temporal Integrity Constraints Using an Active DBMS. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings Publishing Company, 2nd edition, 1994.

[Gad88] S. K. Gadia. A Homogeneous Relational Model and Query Language for Temporal Databases. *ACM Transactions on Database Systems*, 13(4):418–448, December 1988.

[GHR94] D.M. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects.* Oxford University Press, 1994.

[GL93] M. Gertz and U.W. Lipeck. Deriving Integrity Maintaining Triggers from Transition Graphs. In *Proceedings of the International Conference on Data Engineering*, 1993.

[GM91] D. Gabbay and P. McBrien. Temporal Logic and Historical Databases. In *Proceedings of the International Conference on Very Large Databases*, 1991.

[GN93] S. K. Gadia and S. S. Nair. Temporal Databases: A Prelude to Parametric Data. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 28–66. Benjamin/Cummings Publishing Company, 1993.

[JCE+94] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia editors. A Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1):52–64, March 1994.

[Klu82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 29(3):699–717, 1982.

[LM93] T. Y. C. Leung and R. R. Muntz. Stream Processing: Temporal Query Processing and Optimization. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design,* *and Implementation*, chapter 14, pages 329–355. Benjamin/Cummings Publishing Company, 1993.

[LS87] U.W. Lipeck and G. Saake. Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, 12(3):255–269, 1987.

[MS93] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[NA93] S. Navathe and R. Ahmed. Temporal Extensions to the Relational Model and SQL. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 92–109. Benjamin/Cummings Publishing Company, 1993.

[Sar90] N. Sarda. Extensions to SQL for Historical Databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):220–230, June 1990.

[Sar93] N. Sarda. HSQL: A Historical Query Language. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings Publishing Company, 1993.

[Sno87] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Sno95] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 674+xxiv pages, 1995.

[SW95] A.P. Sistla and O. Wolfson. Temporal Triggers in Active Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):471–486, June, 1995.

[Tan86] A. U. Tansel. Adding time dimension to relational model and extending relational algebra. *Information Systems*, 11(4):343–355, 1986.

[Tom95] D. Toman. Point-based vs. Interval-based Temporal Query Languages. TR-CS-95-15, Kansas State University, 1995.

[TN96] D. Toman and D. Niwiński. First-Order Temporal Queries Inexpressible in Temporal Logic. Proc. *EDBT'96* (to appear), 1996.

[TC90] A. Tuzhilin and J. Clifford. A Temporal Relational Algebra as a Basis for Temporal Relational Completeness. In *Proceedings of the International Conference on Very Large Databases*, 1990.

[VGT91] A. Van Gelder and R.W. Topor. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

[Wüt91] B. Wüthrich. *Large Deductive Databases with Constraints.* PhD thesis, Department Informatik, ETH Zürich, 1991.

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