V.4: SQL/TP—a Temporal Extension of SQL

David Toman
University of Waterloo

We use constraint technology as a basis for SQL/TP, a sound *temporal extension* of SQL. In contract to most of the earlier proposals, SQL/TP has clean syntax that is consistent with standard SQL, and unambiguous semantics that is rooted in the theory of constraint databases. We also define a meaningful way of handling aggregation and duplicates, and provide an efficient implementation path on top of existing relational systems based on compilation of queries.

1 Introduction

Research in the area of temporal databases and query languages has produced a large number of proposals for *temporal extensions* of SQL, the standard language of relational databases. However, none of these proposals has been used in commercial database management systems nor become a basis for foundations of temporal database theory. The problems can be traced to overly complex syntax and unclear, ad-hoc, and often poorly defined semantics.

This chapter shows a simple constraint-based approach to managing temporal data that leads to a sound model of temporal databases. This includes a compact representation of temporal relations and a first-order complete query language that retains computational properties of SQL, in particular the data complexity of query evaluation.

The chapter is organized as follows: Section 2 defines a temporal data model based on linear-order constraints that separates the meaning of temporal relations from the actual compact representation in computer storage. Section 3 introduces the extended language SQL/TP. Unlike most other constraint approaches that require specialized constraint engines for evaluating queries, we propose a translation that compiles the extended language to standard SQL and evaluates the result using a standard relational engine. Section 4 completes the picture by introducing the data manipulation part of SQL/TP. We conclude with a few comments on implementation and possible extensions.

2 Temporal Data Model and Concrete Encoding

The underlying temporal data model is based on two sorts:

- a temporal sort T: a discrete, countably infinite integer-like linear order $(Z, \leq)$ to represent individual time instants and the progression of time.
- a data sort D: a set of constants with equality $(\mathbf{dom}, =)$ to stand for all other SQL data types.

The actual structure of the data sort is not important; we disallow interaction between values of different sorts in our model except through user-defined relations. An *abstract temporal database*[1] is a first order structure with two sorts, T and D, extended with user-defined *abstract temporal relations* $R_1, \ldots, R_k$ with attributes ranging over Z and **dom**. To maintain compatibility with SQL, we must allow multisets with finite duplication of elements in place of user-defined relations R_i. This requirement is be balanced with the possibility of potentially infinite abstract temporal relations. In section 3 we introduce simple syntactic restrictions to guarantee closed-form query evaluation.

2.1 Representable Temporal Databases and Concrete Encoding

It is easy to see that storing temporal relations directly leads to enormous space requirements, especially when a fine granularity of time is chosen, and is not possible at all in the case of infinite relations (e.g., for time-invariant facts). This problem has been recognized by most practical approaches to temporal databases and therefore the majority of temporal data models use intervals[2] to represent time in the database. Most of these proposals, however, treat intervals as actual values rather than compact descriptions of contiguous sets of time instants. This leads to semantic problems when defining query languages over such models (cf. example 1 in section 3).

The data model of SQL/TP also uses intervals. However, it recognizes that intervals are only a particular way of representing unary quantifier-free formulas of linear order. More formally, we define a *concrete temporal sort* to be the structure $I(T)$ consisting of the set $I(Z)$ of all closed nonempty intervals in $Z \cup \{-\infty, \infty\}$, equipped with order relations relating endpoints of intervals in the order inherited from T. Each element of $I \in I(T)$ encodes a (satisfiable) formula $\varphi_I(t) = (I^- \leq t \leq I^+)$ (and vice versa) where I^- and I^+ are the left and right endpoints of I. A *concrete temporal database* is a first order structure with two sorts, $I(T)$ and D, extended with finite relations $R_1, \ldots, R_k$ with attributes ranging over $I(Z)$ and **dom**. The correspondence between abstract temporal relations and concrete relations is defined by a semantic map

$$[\![R]\!] = \{(a_1, \ldots, a_k, t_1, \ldots, t_n) : (a_1, \ldots, a_k, I_1, \ldots, I_n) \in R, t_j \in I_j\}$$

similarly to definition 2.4, but taking into account multisets with finite duplication of tuples. The $[\![.]\!]$ map naturally extends to databases, its range defines the class of *finitely representable* abstract temporal databases.

2.2 The Data Definition Language

To declare tables (abstract temporal relations) SQL/TP uses the standard

```
<decl> ::= create table <rid> ( <signature> )
```

[1] Or "unrestricted" in the constraint setting (cf. chapter I.1).

[2] Or other complex values, e.g., temporal elements.

statement where `<signature>` is a list of attribute name-type pairs. Temporal attributes use a new data type `time` that supports the following *encoding modifiers* to determine how sets of time instants are stored in the concrete temporal relation:

`using points`: time instants are stored as atomic values.
`using intervals`: sets of contiguous time instants associated with a particular data tuple are encoded by intervals.

Note that these modifiers affect only the way a table is stored and not its meaning. This is similar to specifying, e.g., a key for a table. The modifiers are the only part of SQL/TP syntax that reflects the chosen encoding.

3 Queries

A common observation made at this point is that *concrete temporal databases* can be viewed as standard *finite* relational databases and thus we can simply use ordinary relational (SQL) queries on top of them. Indeed, this approach has been taken by the majority of temporal query languages (with varying amount of syntactic sugar obscuring this fact). However, this approach leads to serious problems when the semantics of such query languages has to be defined.

Example 1. Consider the query

```
select r.I from R r,R s where r.x=s.x and r.I contains s.I
```

and two concrete databases, D_1 where $R(I,x) = \{([0,3],a),([1,2],a)\}$ and D_2 where $R(I,x) = \{([0,2],a),([1,3],a)\}$. While $[\![D_1]\!] = [\![D_2]\!]$, the query returns $\{[0,3]\}$ for D_1 but $\{\}$ for D_2.

The meaning of the above query is not well defined as it fails to respect the semantic map $[\![.]\!]$ (cf. definition 3.4 in chapter I.1). Recognizing queries that do *not* depend on a particular encoding of databases is extremely hard in these languages. On the other hand, implementation of these approaches is extremely simple: we merely use standard relational technology.

SQL/TP avoids the above problems by defining semantics of queries (and updates) over the abstract temporal databases. Similarly to the more general constraint case, this approach guarantees a well-defined semantics, even if relations may not be finite (cf. definition 3.4 in chapter I.1; we use a few simple restrictions to guarantee this in the presence of duplicates and aggregate operations, similarly to chapter I.4). Semantics defined this way also naturally extends the standard finite case.

3.1 Core Syntax and Semantics

The syntax of the core query language is defined by the following grammar[3]:

[3] The real grammar also defines the precedence of the individual query constructs, etc.

```
<query> ::= <rid>
         |  select [<sexp> as <id> {, <sexp> as <id>}] <query>
         |  from <query> <rid> {, <query> <rid>}
         |  <query> where <condition>
         |  <query> group by <id> {, <id>}
                   [aggregate <cexp> as <id> {, <cexp> as <id>}]
         |  <query> <setop> [all] <query>
```

where `<sexp>` stands for a scalar expression, `<cexp>` for a column expression (an aggregate function), `<condition>` for an atomic condition, and `<setop>` for a multiset operation. For a given temporal database every expression denotes a multiset with attributes (columns) named $R.a$, where R is a correlation name and a an attribute name. Each attribute is associated with a data type determined by the usual typing rules. The meaning of $Q(D)$, for an abstract temporal database D, is given inductively for the individual constructs as follows:

$R(D)$ is the relation stored in D under the name R. The signature is $\{R.a_i : t_i\}$, attribute a_i has type t_i defined in the schema of R.

$(\texttt{select}\, e_1\, \texttt{as}\, i_1, \ldots, e_k\, \texttt{as}\, i_k\ Q)(D) = \{(e_1(x), \ldots, e_k(x)) : x \in Q(D)\}$. Expressions of type `time` are of the form $a_i + c$ where a_i is a temporal attribute and c is a duration of time (a constant). The signature of the result is $\{\text{result}.i_j : t_j\}$ where t_j is the type of the scalar expression e_j.

$(\texttt{from}\, Q_1\ R_1, \ldots, Q_k\ R_k)(D) = Q_1(D) \times \ldots \times Q_k(D)$. The resulting signature is $\{R_i.a_j : t_j\}$ where $P_i.a_j : t_j$ are attributes of Q_i.

$(Q\ \texttt{where}\, \varphi)(D) = \sigma_\varphi Q(D)$. The condition is an atomic formula. For attributes of type `time` of the form $a_i + c \le a_j$, $c \le a_i$, or $a_i \le c$; for data attributes the conditions are defined as in standard SQL. The signature of the result is inherited from Q.

$(Q\ \texttt{group by}\, R_1.a_1, \ldots, R_k.a_k\ \texttt{aggregate}\, f^1(b_1)\, \texttt{as}\, i_1, \ldots, f^k(b_k)\, \texttt{as}\, i_k)(D) = Q(D)\langle a_1, \ldots, a_k; f^1_{b_1}, \ldots, f^k_{b_k}\rangle$ using the notation for aggregation from chapter I.4, definition 2.1: a_i and b_i are attribute names and f^i aggregate functions; for temporal attributes the allowed f^i are `min`, `max`, and `count`. The signature of the result is $\{R_i.a_i : t_i\} \cup \{\text{agg}.i_j : t_j\}$.

$(Q_1 \texttt{<setop>}\ Q_2)(D) = Q_1(D)\ \texttt{op}\ Q_2(D)$. The allowed operations `<setop>` (and the corresponding semantic operators `op`) are `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 Q_1 and Q_2 to match; the result then inherits the common signature.

To guarantee closure of the above constructs we require that the `select` clause may not project out a temporal attribute encoded using intervals (as this might violate the requirement of finite duplication) and the grouped-by and non-grouped attributes in the `group by`—`aggregate` clause are mutually independent whenever aggregate functions `count` and `sum` are used (cf. definition 4.2 in chapter I.4). In addition, to guarantee that the result of a query is representable using the interval encoding we require all attributes in the answer to a query to be pairwise independent. We use a simple abstract interpretation-based technique to approximate attribute independence for query expressions; attributes of base relations are independent due to the chosen interval encoding.

3.2 Syntactic Sugar

The core SQL/TP query language is are sufficient to support most of the additional constructs in standard SQL. The remaining SQL syntax is mapped onto the core language by a simple syntactic manipulation (e.g., `exists` subqueries to `from` subqueries, `not exists` to `except`, `having` and `and` to nested `where`, `or` to `union`, etc.). Queries can also be given names using the usual view mechanism:

```
<decl> ::= create view <rid> ( <query> )
```

The signature of the view is derived from the `<query>` expression.

3.3 Query Compilation

To evaluate queries, most of the constraint-based approaches require specialized query evaluation engine (cf. chapters V.1, V.2, and V.3) usually based on *constraint* counterparts of relational algebra operators (cf. chapter IV.1). We take a different path: our goal is to translate SQL/TP queries to standard SQL and then submit the result to an off-the-shelf database engine. To achieve this goal we have to resolve two problems. First, every temporal attribute t in a SQL/TP query that refers to individual time instants in T has to be replaced with an attribute I ranging over the elements of the concrete temporal sort $I(T)$[4].

Moreover, while the top-level attributes of a SQL/TP query have to be independent, this restriction may not hold for subqueries of the original query. To translate (sub-)queries with dependent attributes we use *conditional queries* of the form $Q\{\varphi\}$ where Q is a SQL translation of the original query (without any references to individual time instants) and φ a formula that captures the remaining attribute dependencies. The translation itself is then realized by a function comp that maps SQL/TP queries (by induction on their structure) to sets of conditional queries while maintaining the following invariant:

$$Q([\![D]\!]) = [\![\mathsf{comp}(Q)(D)]\!] \triangleq \biguplus_{Q^j\{\varphi^j\}\in\mathsf{comp}(Q)} \sigma_{\varphi^j}[\![Q^j(D)]\!]$$

Note that a single query may result in a set of conditional queries. Using the invariant the translations[5] for atomic relations and the `from` clause are as follows:

[4] We assume the existence of an interval-valued data type in the target language; in the absence of such a data type we use pairs of attributes to denote left and right endpoints of intervals instead.

[5] We follow a convention that if Q^j and φ^j appear in the translation, then they range over all j such that $Q^j\{\varphi^j\} \in \mathsf{comp}(Q)$.

$\mathsf{comp}(R) = \{R\{true\}\}$
$\mathsf{comp}(\texttt{from}\, Q_1\ R_1, \ldots, Q_k\ R_k) = \{\texttt{from}\, Q_1^{j_1}\ R_1, \ldots, Q_k^{j_k}\ R_k\{\varphi_1^{j_1} \wedge \ldots \wedge \varphi_k^{j_k}\}\}$

To translate the `where` and `select` clauses we use the following notation: L_φ^t is the set of all maximal lower bounds by interval endpoints and constants for t implied by φ; similarly R_φ^t is the corresponding set of minimal upper bounds:

$$\mathsf{comp}(Q\,\texttt{where}\, t_i \leq t_j) = \{Q^j\,\texttt{where}\, \max(L_{\varphi^j}^{t_i}) \leq \min(R_{\varphi^j}^{t_j})\{\varphi^j \wedge (t_i \leq t_j)\}\}$$

Similarly for constants, with $L_\varphi^c = R_\varphi^c = \{c\}$. We also check if the resulting condition is still satisfiable. This corresponds to composition of the order constraints, including a satisfiability check. (cf. definitions ??? and ??? in chapter II.2). Non-temporal `where` clauses are simply propagated to the translated subqueries. The `select` clause is translated as follows:

$$\mathsf{comp}(\texttt{select}\, e_1\,\texttt{as}\, i_1, \ldots, e_k\,\texttt{as}\, i_k\ Q) = \{\texttt{select}\, e_i'\,\texttt{as}\, i_1, \ldots, e_k'\,\texttt{as}\, i_k\ Q^j\{\psi^j\}\}$$

where (i) temporal expressions $e_i = t + c$ are translated to $e_i' = [\max(L_{\varphi^j}^t) + c, \min(R_{\varphi^j}^t) + c]$, and (ii) the formula φ^j is adjusted to ψ^j by taking into account the temporal expressions e_i.

The second challenge lies in defining multiset and grouping operators that preserve the semantics of queries over the interval-based encoding while maintaining compatibility with SQL. To accomplish this, we introduce a concept of *time compatible queries*: let $Q_1, \ldots, Q_k$ be SQL queries with compatible signatures that contain a temporal attribute I_t and a set of data attributes X. We say that $Q_1, \ldots, Q_k$ are *t-compatible on X* if, for all databases D, whenever a tuple in $Q_i(D)$ and a tuple in $Q_j(D)$ agree on values of X then the value of I_t represents identical or disjoint intervals. Similarly we define compatibility for sets of temporal attributes. The compatibility requirement forces intervals to behave like points with respect to multiset operations and aggregation:

- Let Q_1, Q_2 be time-compatible SQL queries with common signature. Then $[\![Q_1]\!]\ \mathsf{op}\ [\![Q_2]\!] = [\![Q_1\ \mathsf{op}\ Q_2]\!]$ where op is a set operation.
- Let Q be t-compatible on (a subset of) data attributes for all temporal attributes t in the group-by clause. Then $[\![Q]\!]\langle A, f\rangle = [\![Q\langle A, f'\rangle]\!]$ where f' is the translation of the aggregate operator f.

The `group by` clause is also used to eliminate duplicates (necessary to express `select distinct`-like queries) and to project out interval-encoded temporal attributes. In these cases we also use the quantifier elimination procedure for integer order to discard corresponding attributes from the conditional part of the translated query.

To complete the picture, we define a *normalization operation* that transforms a set of arbitrary SQL queries to a set of t-compatible ($[\![.]\!]$-)equivalent queries. Moreover, this operation can be defined using a first-order query:

Lemma 1. *Let $\{Q_1, \ldots, Q_k\}$ be a set of queries with compatible signatures such that X a subset of their data attributes and t a temporal attribute. Then there are first-order queries $\mathsf{N}_X^t[Q_i; Q_1, \ldots, Q_k]$ 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*.*

The N *operator can be extended to handle sets of temporal attributes and can be computed in* $O(n \log n)$ *in the combined size of the results of* $Q_i(D)$*.*

Using the above cases and with the help of DeMorgan laws and a simple induction on the structure of SQL/TP queries we have:

Theorem 1. *Let* Q *be a SQL/TP query. Then* $Q([\![D]\!]) = [\![\mathsf{comp}(Q)(D)]\!]$ *for all concrete databases* D*.*

As the top-level attributes of Q are required to be independent, $\mathsf{comp}(Q)$ returns a pure SQL query. Moreover, the resulting query yields a representable temporal relation when evaluated over a concrete temporal database.

3.4 Optimization and Improvements of the Translation

So far we have presented only a bare-bones translation. Here we simply list several improvements (besides standard relational optimizations):

- the merging of conditional queries with common conditions,
- eliminaton of common subexpressions using views,
- minimization of the number of N operations,
- approximaton of boundedness of intervals,

While not necessary, the above optimizations lead to much more efficient queries, mainly because contemporary commercial query optimizers don't cope well with the results of the translation.

4 Updates

Updates in SQL/TP are handled similarly to multiset operations: the arguments of an update (a base relation and a query) are normalized and then an ordinary SQL update is performed. The only difficulty arises from the need to apply the normalization procedure *on a stored relation*. In the current system we approach this problem by translating each update to a sequence of *normalization updates* that change the concrete encoding while preserving the meaning of the stored relation, followed by the actual update.

4.1 The Data Manipulation Language:

SQL/TP supports the following (simplified) DML syntax:

```
<update> ::= insert into <rid> tuples in <query>
          |  delete from <rid> tuples in <query>
          |  update <rid> set <assgmts> for tuples in <query>
```

Similarly to the query language, the rest of SQL data manipulation statements can be translated to the above syntax. The exception are the *positioned* updates and updates interacting with cursors.

5 Implementation

The translation is implemented as a source-to-source off-line compiler that takes a SQL/TP expression and a description of the database schema and produces an equivalent standard SQL expression (with slight adjustments to the standard to facilitate dialects of particular DBMS). The compiler itself is implemented in JAVA and can, besides emitting standard SQL in textual form, use JDBC as a back end and directly connect to a DBMS.

Bibliographic Notes

A thorough discussion of earlier temporal data models and query languages would require another chapter. A survey [Cho94] discusses many of the individual proposals and compares them in a unifying framework of abstract and concrete temporal databases. The current front-runner for a standard temporal extension of SQL is SQL/Temporal [SBJS96], a successor of TSQL2 [Sno95]. Both of these languages are essentially first-order languages with aggregation over concrete temporal databases, with inherent semantical problems (see below), which prevented its inclusion in the SQL standard.

Coalescing of temporal relations [BSS96], with the implied limitation to a single distinguished *valid time* attribute ranging over intervals (or two distinguished attributes: valid and transaction time), is used to alleviate the semantic problems shown in Example 1. However, restricting temporal relations to a fixed number of temporal attributes severely limits the class of queries that can be implemented using an algebra over relations in such models [AHVdB96,TN96]. Most importantly, it cannot support all first-order queries and thus full SQL[6]. Lifting this limitation, however, precludes the use of coalescing and leads unavoidably to queries without well-defined meaning [CT98,Tom98].

A different approach based on constraints [Tom96] allows evaluation of first-order queries over interval-encoded temporal databases. It also shows that all well defined first-order TSQL2 queries can be formulated using this approach. Unlike [LM97], our approach also guarantees polynomial time data complexity of query evaluation. This set-based approach was extended to multisets with aggregates, including the definition of SQL-like syntax [Tom97].

On the implementation side, besides generic access and indexing methods for constraints (cf. chapter IV.3), many specialized access methods have been developed to account for non-uniform access patterns to temporal data, e.g., for the cases when more current data are accessed more frequently [ST99]. The query processing and optimization for SQL/TP can draw on the rich literature for standard relational queries [Cha98] due to our translation-based approach. However, the translation produces fairly complex SQL queries, which in turn require advanced optimization techniques [FMV94,Ioa96]. In addition, we may have to consider a native implementation of the normalization technique used to evaluate SQL/TP queries, similarly to coalescing [BSS96].

[6] Without resorting to kludges, e.g., to coercing temporal values to integers.

References

[AHVdB96] Serge Abiteboul, Laurent Herr, and Jan Van den Bussche. Temporal Versus First-Order Logic to Query Temporal Databases. In *ACM Symposium on Principles of Database Systems*, pages 49–57, 1996.

[BSS96] Michael Böhlen, Richard T. Snodgrass, and Michael D. Soo. Coalescing in Temporal Databases. In *International Conference on Very Large Data Bases*, pages 180–191, 1996.

[Cha98] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *ACM Symposium on Principles of Database Systems*, pages 34–43, 1998.

[Cho94] Jan Chomicki. Temporal Query Languages: A Survey. In *Temporal Logic, First International Conference*, pages 506–534. Springer-Verlag, LNAI 827, 1994.

[CT98] Jan Chomicki and David Toman. Temporal Logic in Information Systems. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*, chapter 3, pages 31–70. Kluwer, 1998.

[FMV94] Jonathan C. Freytag, David Maier, and Gottfried Vossen. *Query Processing for Advanced Database Systems*. Morgan Kaufman, 1994.

[Ioa96] Yannis E. Ioannidis. Query Optimization. *Computing Surveys*, 28(1):121–123, 1996.

[LM97] Nikos A. Lorentzos and Yannis G. Mitsopoulos. SQL Extensions for Interval Data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):480–499, 1997.

[SBJS96] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. Adding Valid Time to SQL/Temporal. ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2 21/11/96, (change proposal), International Organization for Standardization, 1996.

[Sno95] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[ST99] Betty Salzberg and Vassilis J. Tsotras. A Comparison of Access Methods for Temporal Data. *ACM Computing Surveys*, 1999. to appear.

[TN96] David Toman and Damian Niwinski. First-order queries over temporal databases inexpressible in temporal logic. In *International Conference on Extending Database Technology*, pages 307–324, 1996.

[Tom96] David Toman. Point vs. Interval-based Query Languages for Temporal Databases. In *ACM Symposium on Principles of Database Systems*, pages 58–67, 1996.

[Tom97] David Toman. Point-based Temporal Extensions of SQL. In *International Conference on Deductive and Object-Oriented Databases*, pages 103–121, 1997.

[Tom98] David Toman. Point-Based Temporal Extensions of SQL and Their Efficient Implementation. In Opher Etzion, Sushil Jajodia, and Suryanarayana Sripada, editors, *Temporal Databases: Research and Practice*, pages 211–237. Springer LNCS State-of-the-Art Survey, 1998.

Index

aggregate operator, 4
– closure of, 4

complexity
– data, 1
constraints
– integer order, 1
– – unary, 2
constraint encoding
– modifiers, 3

database
– finitely representable, 2
– temporal, 1
data model
– temporal, 1

finite duplication, 2, 4

interval
– as encoding of order constraint, 2
– as value, 2

language
– data definition, 2
– data manipulation, 7

multiset, 2

quantifier elimination
– integer order, 6
query
– evaluation engine, 5
– translation to SQL, 5
query language
– first-order complete, 1
– temporal, 1

SQL
– temporal extension of, 1

temporal
– database, 1
– – abstract, 2, 8
– – concrete, 2, 8
– – encoding using intervals, 2
– data model, 1
– query
– – conditional, 5
– – data complexity, 1
– – normalization operator, 7
– – time compatible, 6

update, 7

variable independence
– closure for aggregation, 4
view, 5