Abstract Versus Concrete Temporal Query Languages

Jan Chomicki, University at Buffalo, USA, `http://www.cse.buffalo.edu/~chomicki`
David Toman, University of Waterloo, Canada, `http://www.cs.uwaterloo.ca/~david`

SYNONYMS

historical query languages

DEFINITION

Temporal query languages are a family of query languages designed to query (and access in general) time-dependent information stored in temporal databases. The languages are commonly defined as extensions of standard query languages for non-temporal databases with *temporal features*. The additional features reflect the way dependencies of data on time are captured by and represented in the underlying temporal data model.

HISTORICAL BACKGROUND

Most databases store time-varying information. On the other hand, SQL is often the language of choice for developing applications that utilize the information in these databases. Plain SQL, however, does not seem to provide adequate support for temporal applications.

Example. To represent the *employment histories* of persons, a common relational design would use a schema

$$\texttt{Employment}(\texttt{FromDate}, \texttt{ToDate}, \texttt{EID}, \texttt{Company}),$$

with the intended meaning that a person identified by `EID` worked for `Company` continuously from `FromDate` to `ToDate`. Note that while the above schema is a standard relational schema, the additional assumption that the values of the attributes `FromDate` and `ToDate` represent *continuous periods* of time is itself *not* a part of the relational model.

Formulating even simple queries over such a schema is non-trivial: for example the query GAPS: *"List all persons with gaps in their employment history, together with the gaps"* leads to a rather complex formulation in, e.g., SQL over the above schema (this is left as a challenge to readers who consider themselves SQL experts; for a list of appealing, but incorrect solutions, including the reasons why, see [9]). The difficulty arises because a single tuple in the relation is conceptually a *compact representation of a set of tuples*, each tuple stating that an employment fact was true on a particular day.

The tension between the conceptual abstract temporal data model (in the example, the property that employment facts are associated with individual *time instants*) and the need for an efficient and compact representation of temporal data (in the example, the representation of continuous periods by their start and end instants) has been reflected in the development of numerous temporal data models and temporal query languages [3].

SCIENTIFIC FUNDAMENTALS

Temporal query languages are commonly defined using *temporal extensions* of existing non-temporal query languages, such as relational calculus, relational algebra, or SQL. The temporal extensions can be categorized in two, mostly orthogonal, ways:

The choice of the actual temporal values manipulated by the language. This choice is primarily determined by the underlying temporal data model. The model also determines the associated operations on these values. The meaning of temporal queries is then defined in terms of temporal values and operations on them, and their interactions with *data* (non-temporal) values in a temporal database.

The choice of syntactic constructs to manipulate temporal values in the language. This distinction determines whether the temporal values in the language are accessed and manipulated *explicitly*, in a way similar to other values stored in the database, or whether the access is *implicit*, based primarily on *temporally extending* the meaning of constructs that already exist in the underlying non-temporal language (while still using the operations defined by the temporal data model).

Additional design considerations relate to *compatibility* with existing query languages, e.g., the notion of temporal upward compatibility.

However, as illustrated above, an additional hurdle stems from the fact that many (early) temporal query languages allowed the users to manipulate a *finite underlying representation* of temporal databases rather than the actual temporal values/objects in the associated temporal data model. A typical example of this situation would be an approach in which the temporal data model is based on time instants, while the query language introduces interval-valued attributes. Such a discrepancy often leads to a complex and unintuitive semantics of queries.

In order to clarify this issue, Chomicki has introduced the notions of *abstract* and *concrete* temporal databases and query languages [2]. Intuitively, *abstract temporal query languages* are defined at the conceptual level of the temporal data model, while their *concrete* counterparts operate directly on an actual *compact encoding* of temporal databases. The relationship between abstract and concrete temporal query languages is also implicitly present in the notion of snapshot equivalence [7]. Moreover, Bettini *et al.* [1] proposed to distinguish between *explicit* and *implicit* information in a temporal database. The explicit information is stored in the database and used to derive the implicit information through *semantic assumptions*. Semantic assumptions about fact persistence play a role similar to mappings between concrete and abstract databases, while other assumptions are used to address time-granularity issues.

Abstract Temporal Query Languages

Most temporal query languages derived by temporally extending the relational calculus can be classified as abstract temporal query languages. Their semantics is defined in terms of abstract temporal databases which, in turn, are typically defined within the point-stamped temporal data model, in particular *without* any additional hidden assumptions about the meaning of tuples in instances of temporal relations.

Example. The *employment histories* in an abstract temporal data model would most likely be captured by a simpler schema "$\texttt{Employment}(\texttt{Date}, \texttt{EID}, \texttt{Company})$", with the intended meaning that a person identified by `EID` was working for `Company` on a particular `Date`. While instances of such a schema can be potentially very large (especially when a fine granularity of time is used), formulating queries is now much more natural.

Choosing abstract temporal query languages over concrete ones resolves the first design issue: the temporal values used by the former languages are time instants equipped with an appropriate temporal ordering (which is typically a linear order over the instants), and possibly other predicates such as temporal distance. The second design issue—access to temporal values—may be resolved in two different ways, as exemplified by the following two different query languages:

- Temporal Relational Calculus (TRC): a two-sorted first-order logic with variables and quantifiers explicitly ranging over the time and data domains (see the entry Temporal Relational Calculus).
- First-order Temporal Logic (FOTL): a language with an implicit access to timestamps using temporal connectives (see the entry Temporal Logic in Database Query Languages).

Example. The GAPS query is formulated as follows:

TRC: $\exists t_1, t_3. t_1 < t_2 < t_3 \land \exists c.\texttt{Employment}(t_1, x, c) \land (\neg\exists c.\texttt{Employment}(t_2, x, c)) \land \exists c.\texttt{Employment}(t_3, x, c)$

FOTL: $\blacklozenge \exists c.\texttt{Employment}(x, c) \land (\neg\exists c.\texttt{Employment}(x, c)) \land \lozenge\exists c.\texttt{Employment}(x, c)$

Here, the explicit access to temporal values (in TRC) using the variables t_1, t_2, and t_3 can be contrasted with the implicit access (in FOTL) using the temporal operators $\blacklozenge$ (read "sometime in the past") and $\lozenge$ (read "sometime in the future"). The conjunction in the FOTL query represents an implicit temporal join. The formulation in

TRC leads immediately to an equivalent way of expressing the query in SQL/TP [9], an extension of SQL based on TRC (see the entry SQL-based Temporal Query Languages).

Example. The above query can be formulated in SQL/TP as follows:

```
SELECT  t.Date, e1.EID
FROM    Employment e1, Time t, Employment e2
WHERE   e1.EID = e2.EID AND e1.Date < e2.Date
  AND NOT EXISTS ( SELECT *
                   FROM   Employment e3
                   WHERE  e1.EID = e3.EID   AND t.Date = e3.Date
                     AND  e1.Date < e3.Date AND e3.Date < e2.Date )
```

The unary constant relation `Time` contains all time instants in the time domain (in our case, all `Dates`) and is only needed to fulfill syntactic SQL-style requirements on attribute ranges. However, despite of the fact that the instance of this relation is not finite, the query can be efficiently evaluated [9].

Note also that in all the above cases, the formulation is *exactly the same* as if the underlying temporal database used the *plain* relational model (allowing for attributes ranging over time instants).

The two languages, FOTL and TRC, are the counterparts of the snapshot and timestamp models (cf. the entry Point-stamped Data Models) and are the roots of many other temporal query languages, ranging from the more TRC-like temporal extensions of SQL, to more FOTL-like temporal relational algebras (e.g., the conjunction in temporal logic directly corresponds to a temporal join in a temporal relational algebra, as both of them induce an *implicit equality* on the associated time attributes). The precise relationship between these two groups of languages is investigated in the entry Temporal Logic in Database Query Languages.

Temporal integrity constraints over point-stamped temporal databases can also be conveniently expressed in TRC or FOTL (see the entry Temporal Integrity Constraints).

Multiple Temporal Dimensions and Complex Values. While the abstract temporal query languages are typically defined in terms of the point-based temporal data model, they can similarly be defined with respect to complex temporal values, e.g., pairs (or tuples) of time instants or even sets of time instants. In these cases, in particular in the case of set-valued attributes, it is important to remember that the set values are treated as *indivisible objects*, and hence truth (i.e., query semantics) is associated with the entire objects, but not necessarily with their components/subparts. For a detailed discussion of this issue, see the entry Telic Distinction in Temporal Databases.

Concrete Temporal Query Languages

Although abstract temporal query languages provide a convenient and clean way of specifying queries, they are not immediately amenable to implementation: the main problem is that, in practice, in temporal databases facts persist over periods of time. Storing all true facts individually *for every time instant* during a period would be prohibitively expensive or, in the case of infinite time domains such as *dense time*, even impossible.

Concrete temporal query languages avoid these problems by operating directly on the compact encodings of temporal databases (see the discussion of compact encodings in the entry on Point-stamped Temporal Models). The most commonly used encoding is the one that uses *intervals*. However, in this setting, a tuple that associates a fact with such an interval is a compact representation of the association between the same fact and *all the time instants that belong to this interval.* This observation leads to the design choices that are commonly present in such languages:

- Coalescing is used, explicitly or implicitly, to consolidate representations of (sets of) time instants associated *with the same fact.* In the case of interval-based encodings, this leads to coalescing adjoining or overlapping intervals into a single interval (see the entry Temporal Coalescing). Note that coalescing only changes the *concrete representation* of a temporal relation, not its meaning (i.e., the abstract temporal relation); hence it has no counterpart in abstract temporal query languages.

- Implicit *set operations* on time values are used in relational operations. For example, conjunction (join) typically uses set intersection to generate a compact representation of the time instants attached to the facts in the result of such an operation.

Example. For the running example, a concrete schema for the employment histories would typically be defined as "`Employment(VT, EID, Company)`", where `VT` is a valid time attribute ranging over periods (intervals). The GAPS query can be formulated in a calculus-style language corresponding to TSQL2 (see the entry on TSQL2) along the following lines:

$$\exists I_1, I_2.\, [\exists c.\texttt{Employment}(I_1, x, c)] \wedge [\exists c.\texttt{Employment}(I_2, x, c)] \wedge I_1 \ \texttt{precedes}\ I_2 \wedge I = [\mathsf{end}(I_1)+1, \mathsf{begin}(I_2)-1].$$

In particular, the variables I_1 and I_2 range over periods and the `precedes` relationship is one of Allen's interval relationships. The final conjunct, $I = [\mathsf{end}(I_1)+1, \mathsf{begin}(I_2)-1]$, creates a new period corresponding to the time instants related to a person's *gap in employment*; this interval value is explicitly constructed from the end and start points of I_1 and I_2, respectively. For the query to be correct, however, the results of evaluating the bracketed subexpressions, e.g., "$[\exists c.\texttt{Employmeent}(I_1, x, c)]$," have to to be *coalesced.* Without the insertion of the explicit coalescing operators, the query is *incorrect.* To see that, consider a situation in which a person p_0 is first employed by a company c_1, then by c_2, and finally by c_3, without any gaps in employment. Then without coalescing of the bracketed subexpressions of the above query, p_0 will be returned as a part of the result of the query, which is incorrect. Note also that it is not enough for the underlying (concrete) database to be coalesced.

The need for an explicit use of coalescing makes often the formulation of queries in some concrete SQL-based temporal query languages cumbersome and error-prone.

An orthogonal issue is the difference between explicit and implicit access to temporal values: this distinction carries over to the concrete temporal languages as well. Typically, the various temporal extensions of SQL are based on the assumption of an explicit access to temporal values (often employing a built-in *valid time* attribute ranging over intervals or temporal elements), while many temporal relational algebras have chosen to use the implicit access based on temporally extending standard relational operators such as temporal join or temporal projection.

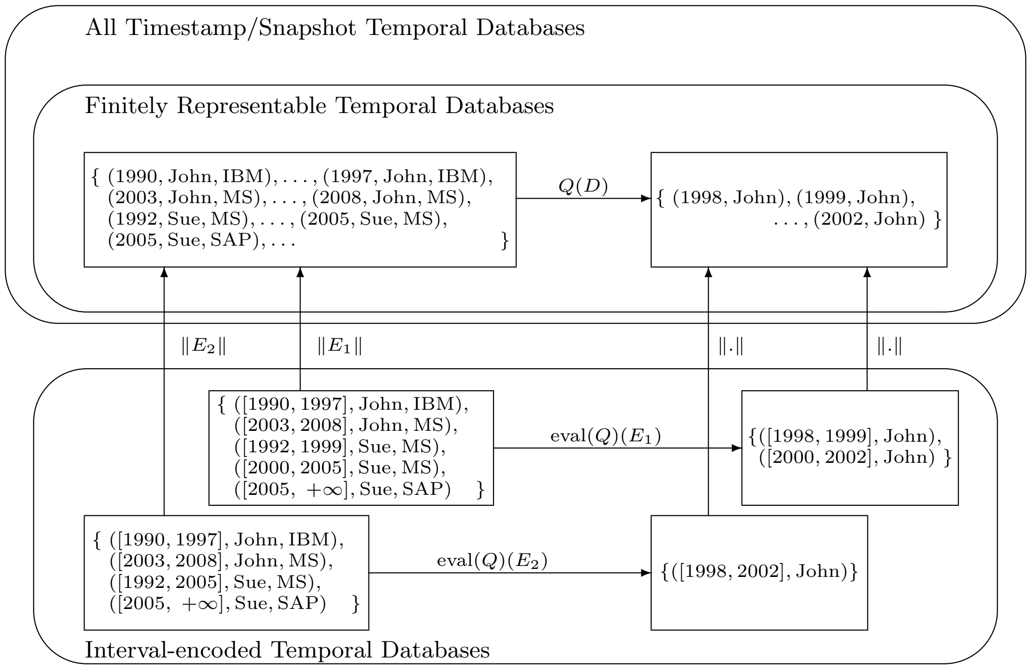
All Timestamp/Snapshot Temporal Databases
Finitely Representable Temporal Databases
{ (1990, John, IBM), . . . , (1997, John, IBM), (2003, John, MS), . . . , (2008, John, MS), (1992, Sue, MS), . . . , (2005, Sue, MS), (2005, Sue, SAP), . . . }
Q(D)
{ (1998, John), (1999, John), . . . , (2002, John) }
‖E₂‖
‖E₁‖
‖·‖
‖·‖
{ ([1990, 1997], John, IBM), ([2003, 2008], John, MS), ([1992, 1999], Sue, MS), ([2000, 2005], Sue, MS), ([2005, +∞], Sue, SAP) }
eval(Q)(E₁)
{([1998, 1999], John), ([2000, 2002], John) }
{ ([1990, 1997], John, IBM), ([2003, 2008], John, MS), ([1992, 2005], Sue, MS), ([2005, +∞], Sue, SAP) }
eval(Q)(E₂)
{([1998, 2002], John)}
Interval-encoded Temporal Databases

Figure 1: Query Evaluation over Interval Encodings of Point-stamped Temporal Databases

Compilation and Query Evaluation. An alternative to allowing users direct access to the encodings of temporal databases is to develop techniques that allow the evaluation of *abstract temporal queries* over these encodings. The main approaches are based on *query compilation* techniques that map abstract queries to concrete queries, while preserving query answers. More formally:

$$Q(\|E\|) = \|\mathsf{eval}(Q)(E)\|,$$

where Q an abstract query, $\mathsf{eval}(Q)$ the corresponding concrete query, E is a concrete temporal database, and $\|.\|$ a mapping that associates encodings (concrete temporal databases) with their abstract counterparts (cf. Figure 1). Note that a single abstract temporal database, D, can be encoded using several *different* instances of the corresponding concrete database, e.g., E_1 and E_2 in Figure 1.

Most of the practical temporal data models adopt a common approach to physical representation of temporal databases: with every fact (usually represented as a tuple), a *concise encoding* of the set of time points at which the fact holds is associated. The encoding is commonly realized by *intervals* [6, 7] or temporal elements (finite unions of intervals). For such an encoding it has been shown that both First-Order Temporal Logic [4] and Temporal Relational Calculus [8] queries can be *compiled* to first-order queries over a natural relational representation of the interval encoding of the database. Evaluating the resulting queries yields the interval encodings of the answers to the original queries, as if the queries were directly evaluated on the point-stamped temporal database. Similar results can be obtained for more complex encodings, e.g., periodic sets, and for abstract temporal query languages that adopt the duplicate semantics matching the SQL standard, such as SQL/TP [9].

KEY APPLICATIONS

Temporal query languages are primarily used for querying temporal databases. However, because of their generality they can be applied in other contexts as well, e.g., as an underlying conceptual foundation for querying sequences and data streams [5].

CROSS REFERENCE

Allen's relations, bitemporal relation, constraint databases, key, nested relational model, non first normal form (N1NF), point-stamped temporal models, relational model, snapshot equivalence, SQL, telic distinction in temporal databases, temporal coalescing, temporal data models, temporal element, temporal granularity, temporal integrity constraints, temporal join, temporal logic in database query languages, temporal relational calculus and algebra, time domain, time instant, TSQL2, transaction time, valid time.

RECOMMENDED READING

[1] C. Bettini, X. S. Wang, and S. Jajodia. Temporal Semantic Assumptions and Their Use in Databases. *Knowledge and Data Engineering*, 10(2):277–296, 1998.

[2] J. Chomicki. Temporal Query Languages: A Survey. In D. Gabbay and H. Ohlbach, editors, *Temporal Logic, First International Conference*, pages 506–534. Springer-Verlag, LNAI 827, 1994.

[3] J. Chomicki and D. Toman. Temporal Databases. In M. Fischer, D. Gabbay, and L. Villa, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*, pages 429–467. Elsevier *Foundations of Artificial Intelligence*, 2005.

[4] J. Chomicki, D. Toman, and M. H. Böhlen. Querying ATSQL Databases with Temporal Logic. *ACM Transactions on Database Systems*, 26(2):145–178, 2001.

[5] Y.-N. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *International Conference on Very Large Data Bases*, pages 492–503, 2004.

[6] S. B. 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, 1993.

[7] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM Trans. Database Syst.*, 12(2):247–298, 1987.

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

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