# 28

# Temporal Statement Modifiers

**Michael H. Böhlen and Christian S. Jensen**

A wide range of database applications manage time-varying data. Although temporal database technology has reached a level of maturity and sophistication where it is evident that these applications may benefit substantially from built-in temporal support in the database management system (DBMS), these applications typically run in an ad-hoc fashion on top of conventional relational systems. This state of affairs may be explained by the fact that it is not clear how to smoothly migrate from a non-temporal DBMS to one that provides systematic and comprehensive temporal support.

The topic of what requirements the data model and query language of a temporal DBMS should satisfy to facilitate this transition has only received scant attention so far, but is nonetheless vital in order to unlock the potential of temporal technology for application in practice. This paper defines a set of such requirements.

The paper introduces the notion of *statement modifiers* that provide a means of systematically adding comprehensive temporal support to an existing data model and query language. Statement modifiers apply to all query language statements, e.g., joins, set difference, subqueries, aggregation, integrity constraints, assertions, views, and data manipulation statements. It is demonstrated how to temporally extend SQL–92 with statement modifiers while fulfilling each of the requirements. A temporally extended SQL–92 is formally defined via a denotational-semantics-style mapping of temporal statements to expressions using a combination of temporal relational and relational algebraic operators. A prototype implementation of the temporal SQL–92 extension is accessible online.

## 1 Introduction

A wide variety of applications manage substantial amounts of time-varying data. They include financial applications such as portfolio management, budgeting, accounting, and banking; record-keeping applications, such as personnel, medical-record, insurance policies, and inventory; and they include travel applications such as airline, train, and hotel reservations and schedule management. Thus, numerous database applications manage substantial quantities of time-varying data. This has held true for as long as databases have been maintained [43, 37]. Along with the continued improvement of storage technologies and new, data-intensive applications such as decision support and data warehousing, old versions of data are retained longer in the databases. This yields very large databases with all data exhibiting a prominent temporal dimension.

In stark contrast, conventional relational database technology provides only little support for temporal data management and is incapable of exploiting the time dimension to achieve better performance. In response to this unfulfilled potential for improvement, much work on temporal database management has been conducted over the past decade or two, leading to, e.g., a wide variety of data models and query languages and to numerous performance-enhancing implementation techniques. Recent query languages (e.g., IXSQL [21], TempSQL [15], and TSQL2 [39]) demonstrate that temporal application development may benefit substantially from built-in temporal support in the query language.

However, whether or not temporal database technology will gain wide acceptance in practice is not only determined by the availability and quality of temporal languages and features galore, but also (and perhaps mainly!) by the ease of transitioning from the existing technology. Despite its importance, this issue has never been discussed thoroughly.

We present a comprehensive set of requirements that not only takes migration concerns into consideration, but also ensures systematic and comprehensive built-in temporal support. The requirement of *upward compatibility* guarantees that replacing the existing DBMS with a new, temporal DBMS does not affect the functioning of any application code. When new code is developed and legacy code is revised, it becomes important that new and old code can coexist harmoniously. This is ensured by the *temporal upward compatibility* requirement. The reuse of existing programmer expertise for the formulation of advanced temporal statements is also desirable. We argue that a temporal DBMS should be a *syntactically similar snapshot reducible* extension of a nontemporal DBMS. This ensures comprehensive built-in support for the point-based view of a temporal database, which is natural for many applications. Beyond this point-based view, the specific timestamping of database facts with intervals is also of importance for some applications. The requirement of *interval preservation* guarantees that query language statments with built-in temporal semantics respect the intervals of their arguments in a specific technical sense. Finally, the *non-restrictiveness* requirement provides the ability to, whenever convenient, override the built-in temporal semantics and instead manipulate time explicitly.

When developing a general-purpose temporal data model and query language, two temporal aspects of data attract special attention. The *valid time* (vt) of a database fact (e.g., a tuple) is the times when the fact was or will be true in the modeled reality. The *transaction time* (tt) of a database fact is the times when the fact has been stored as current in the database. All database facts have a valid time and a transaction time, and we consider both of these times in this paper. There is no requirement that a database captures either of these aspects. We will use the modifiers *temporal* or *time-varying* for databases if one or both of valid and transaction time are associated with their facts. For more specific situations, we use the terms valid-time database for a database that records valid time only, transaction-time database for a database that records transaction time only, and bitemporal database for a database that records both valid and transaction time.

The paper shows how so-called *temporal statement modifiers* may be employed to design a systematic and comprehensive temporal extension of SQL that satisfies the requirements. In addition to queries (select statements), we also address data definition statements, modification statements, and integrity constraints. The language is described in two steps. First, it is shown how the requirements shape the skeleton of a language. Second, a formal definition of the semantics of the query language is provided by means of a denotational-semantics-style mapping to well-defined algebraic expressions. This mapping assumes a mapping of SQL–92 to relational algebra and defines temporal statements in terms of their mapping to well-defined relational and temporal relational algebra expressions. The temporal relational algebra used here is efficiently implementable in that the evaluation of its expressions relies only on the end points of periods and not on intermediate points, making evaluation granularity independent.

Having defined the temporal extension based on statement modifiers, its properties are subjected to scrutiny. First, it is shown that its definition indeed satisfy the migration-related requirements. Second, additional properties of the language are considered. In particular, we compare semantic defaults as provided by statement modifiers with syntactic defaults as chosen by most other temporal languages.

Tiger, a prototype system that implements the temporal SQL is accessible online, via URL <`http://www.cs.auc.dk/~tigeradm`>.

The paper is structured as follows. The next section is concerned with the formulation of the requirements to a temporal data model that, when satisfied, will guarantee a smooth transition of legacy applications from a non-temporal DBMS to a temporally enhanced DBMS with adanced temporal support. Section 3 then proceeds by illustrating how a sample language, SQL–92, may be systematically extended with statement modifiers to provide built-in support for temporal database management. Having provided the rationale and intuition behind the language design, Section 4 gives a concise, yet precise and comprehensive semantics for the language. This provides a solid footing for the exploration of language properties—the topic of Section 5. Related research is explored in some detail in Section 6, and Section 7 summarizes and points to opportunities for future research. Appendices with detailed technical matter complete the paper.

## 2 The Smooth Migration to a Temporal DBMS

Initially, an overview and a description of the assumed context is given. Subsequent sections explore problems that may occur when migrating database applications from an existing to a new temporal DBMS, and they precisely formulate a number of requirements to the temporal DBMS that must be satisfied to facilitate a smooth migration. Throughout, we use simple SQL-based examples, with new constructs underlined, for illustration purposes. This section assumes only a working knowledge of SQL–92. Sections 3 and 4 properly define the syntax and semantics of the examples.

### 2.1 Overview and Context

The prospective users of temporal database technology are enterprises with applications that manage potentially large amounts of time-varying data. These applications may benefit substantially from built-in temporal support in the DBMS. Temporal queries that are shorter and more easily formulated are among the potential benefits. This leads to improved productivity, correctness, and maintainability. It is also a matter of fact that these enterprises are already managing time-varying data, with the applications already in place and working. Indeed, the uninterrupted operation of the existing applications is likely to be of vital importance to any enterprise. The question is then how the enterprise can smoothly migrate from its current DBMS to a temporal DBMS.

We assume that the interface of a DBMS is captured in a data model and thus talk about the migration of application code using an existing data model to using a new data model. We will adopt the convention that a data model consists of two components, namely a set of data structures and a language for querying the data structures [41]. For example, the central data structure of the relational model is the relation, and the central, user-level query language is SQL.

Notationally, $M = (DS, QL)$ then denotes a data model $M$ consisting of a data structure component $DS$ and a query language component $QL$. Thus, $DS$ is the set of all databases, schemas, and associated instances expressible by $M$, and $QL$ is the set of all update and query statements in $M$ that may be applied to some database in *DS*. We use $db$ to denote a database; a statement is denoted by $s$ and is either a query $q$ or an update $u$ (in SQL–92, any modification statement, i.e., `INSERT`, `DELETE`, or `UPDATE`).

## 2.2 Upward Compatibility

It is of fundamental importance to ensure that all code without modification will work with the new system, exactly with the same functionality as with the existing system. The next two definitions capture the essence of what is needed for that to be possible.

We define a data model to be *syntactically upward compatible* with another data model if all the data structures and legal query expressions of the latter model are contained in the former model.

**Definition 1 (syntactical upward compatibility)** Let $M_1 = (DS_1, QL_1)$ and $M_2 = (DS_2, QL_2)$ be two data models. Model $M_1$ is *syntactically upward compatible* with model $M_2$ iff

- $\forall db_2 \in DS_2 \, (db_2 \in DS_1)$ and
- $\forall s_2 \in QL_2 \, (s_2 \in QL_1)$. □

When transitioning from one system to a new system, it is important that the new data model contains the existing data model. If that is the case, all existing application code will remain syntactically correct.

For a query language expression $s$ and an associated database $db$, both legal elements of $QL$ and $DS$ of data model $M = (DS, QL)$, we let $\langle\!\langle s(db) \rangle\!\rangle_M$ be the result of applying $s$ to $db$ in data model $M$. With this notation, we can precisely describe the requirements to a new model that guarantee uninterrupted operation of all application code. In addition to the previous syntactical requirement, we add the requirement that all queries expressible in the existing model must evaluate to the same results in the existing and new models.

**Definition 2 (upward compatibility)** Let $M_1 = (DS_1, QL_1)$ and $M_2 = (DS_2, QL_2)$ be two data models. Model $M_1$ is *upward compatible* with model $M_2$ iff

- $M_1$ is syntactically upward compatible with $M_2$, and
- $\forall db_2 \in DS_2 \, (\forall s_2 \in QL_2 \, (\langle\!\langle s_2(db_2) \rangle\!\rangle_{M_2} = \langle\!\langle s_2(db_2) \rangle\!\rangle_{M_1}))$. □

The first condition implies that all existing databases and query expressions in the old system are also legal databases in the new system. The second condition guarantees that all existing queries compute the same results in the new system as in the old system. Thus, the bulk of legacy application code is not affected by the transition to a new system.

To illustrate upward compatibility (UC), consider the following statements.

```
CREATE TABLE p (A INTEGER CONSTRAINT p_pk PRIMARY KEY);
CREATE TABLE q (B INTEGER, C INTEGER,
                FOREIGN KEY (B) REFERENCES p(A));
CREATE VIEW v AS
SELECT * FROM p WHERE A NOT IN (SELECT B FROM q);
SELECT AVG(B), C FROM q GROUP BY C;
DELETE FROM p WHERE NOT EXISTS (SELECT * FROM q WHERE B > A);
```

These statements are simple legacy SQL–92 statements that must be supported by any reasonable (temporal) extension of SQL–92. The semantics is the one dictated by SQL–92 [23]. The first data definition statement defines a table with one column. A column constraint is used to state that the column is a primary key. The second statement defines a table with two columns. A table constraint is used to make `q.B` reference `p.A`. The third statement defines a view that returns all tuples in `p` that are not referenced from `q`. The select statement that follows groups table `q` on the `C` column and determines the average `B`-value for each group. The last statement deletes all rows in `p` with a `A`-value smaller than the smallest value of `B` in `q`.

By requiring that a temporal extension is a strict superset (i.e., only *adding non-mandatory* constructs and semantics), it is relatively easy to ensure that the temporal extension is upward compatible with SQL–92. Still, it should be noted that upward compatibility does place strict constraints on the temporal extension: It must be “in the spirit” of and must live with any peculiarities of the language it extends. As an example, when extending SQL–92 with a data type for intervals, the string “interval” cannot be used in the syntax because this string is already used for the data type of durations.[1]

There is one unintended ramification of the upward compatibility definition. Any temporal extension that includes new reserved keywords will violate upward compatibility. The reason is that legacy query language statements may have employed such keywords as identifiers. Under the semantics of the new model, such statements will be illegal. However, it is impractical to exclude new reserved keywords from a temporal as well as non-temporal extension. For example, SQL–92 added some 112 reserved keywords to the 115 reserved keywords of its predecessor, SQL–89[2]. One proposed solution is to use *quoted* identifiers in legacy code where identifiers conflict with new reserved keywords and in new code, to avoid future problems. In conclusion, to follow current practice and to avoid being overly restrictive, we consider upward compatibility to be satisfied even when new keywords are added in the extension.

[1]This is the reason why we generally use the SQL3 term “period” for intervals.

[2]Reference [23] provides a list of 10 items with incompatibilities among SQL–89 and SQL–92, with the keyword aspect being one item.

## 2.3 Temporal Upward Compatibility

While essential, upward compatibility is only a first step. Upon adopting a temporal data model, the benefits of the built-in temporal support are only realized incrementally, by modifying existing application code or developing new application code that exploits the temporal capabilities. A next step is thus to formulate requirements that aim at ensuring a harmonious coexistence of legacy application code and new, temporally-enhanced application code.

To see the point of tension between the two, assume that the new temporal model is in place. No application code has been modified, and all relations are thus snapshot relations. Now, an application needs support for the temporal dimension of the data in one of the existing relations. To accommodate this need, the existing snapshot relation is changed to become a temporal relation. It is undesirable to have to change the (legacy) application code that accesses the snapshot relation that has become temporal. It may even be that the source code of the legacy application is no longer available. Therefore, we formulate a requirement stating that the existing applications on snapshot relations must continue to work unmodified when the relations are altered to become temporal. Intuitively, the requirement is that a query $q$ must return the same result on an associated snapshot database $db$ as on the temporal counterpart of the database, $\mathcal{T}(db)$. Further, updates should not affect this.

**Definition 3 (temporal upward compatibility)** Let a temporal and a snapshot data model be given by $M_T = (DS_T, QL_T)$ and $M_S = (DS_S, QL_S)$, respectively. Also, let $\mathcal{T}$ be an operator that changes the type of a snapshot relation to the temporal relation with the same explicit attributes. Next, let $\mathcal{U} = u_1, u_2, \ldots, u_n$ $(n \geq 0)$ denote a sequence of update operations. With these definitions, model $M_T$ is *temporal upward compatible* with model $M_S$ iff

- $M_T$ is upward compatible with $M_S$ and
- $\forall db_S \in DS_S\ (\forall \mathcal{U}\ (\forall q_S \in QL_S\ (\langle\langle q_S(\mathcal{U}(db_S))\rangle\rangle_{M_S} = (\langle\langle q_S(\mathcal{U}(\mathcal{T}(db_S)))\rangle\rangle_{M_T})))).$ □

The subset of the functionality of a temporal data model that corresponds to temporal upward compatibility (TUC) consists of all SQL–92 language constructs, the means of creating temporal tables, and the ability of applying SQL–92 queries and updates to temporal tables. To illustrate, we assume that a temporal DBMS satisfying TUC is in place and build on the statements from the previous section.

```
ALTER TABLE p ADD VT;
INSERT INTO p VALUES (6);
DELETE FROM p VALUES (3);
SELECT * FROM p WHERE NOT EXISTS (SELECT * FROM q
                                  WHERE q.B = p.A);
SELECT * FROM v;
```

The first statement extends `p` to capture valid time by making it a valid-time table. (Alternatively, valid time could be captured by adding a special valid-time column, thus not altering the table type [6].) The other statements are all legacy SQL–92 statements, but their semantics have changed because the underlying data structure, i.e., table `p`, has changed. For example, the modification statements must adequately maintain the valid time of `p` [3]. Also, integrity checking must take into consideration that `p` is temporal whereas `q` still is a snapshot table. Specifically, `p` must be restricted to the current state. The same holds true for the fourth statement. Finally, querying the view requires that the semantics of the view has been redefined (which would be achieved in practice by recompiling the view definition).

**Example 1** Let `p` be $\{\langle 1\|5–8\rangle, \langle 3\|1–NOW\rangle, \langle 4\|1–5\rangle\}$ (the "$\|$" separates the valid time from explicit attributes), let `q` be $\{\langle 1, 2\rangle, \langle 3, 4\rangle\}$, and let the current time be 7. Assume the data definitions from the previous section and the queries just given.

- The database is consistent because, at time 7, both integrity constraints are satisfied, i.e., `p.A` is a primary key and `q.B` references `p.A`.
- Adding $\langle 1\|2-6\rangle$ to `p` leaves the database in a consistent state. Because of TUC, the integrity constraints are checked on the state that is valid at time 7 only.
- Adding either $\langle 1\|6-9\rangle$ to `p` or $\langle 2, 4\rangle$ to `q` violates the consistency of the database. In the first case `p.A` is no longer a primary key at time 7; in the second case, the referential integrity from `q.B` to `p.A` is violated at time 7.
- The insert statement adds $\langle 6\|7-NOW\rangle$ to `p`.
- The delete statement changes $\langle 3\|1-NOW\rangle$ to $\langle 3\|1-6\rangle$ (we assume closed intervals).
- Each of the two queries returns an empty table because all values of `p.A` occur in `q.B` as well at time 7. □

In summary, with temporal upward compatibility, existing applications containing both queries and update statements are not affected when relations are made temporal. New applications that use temporal relations may thus coexist with existing applications, making it feasible for new applications to take incrementally advantage of the built-in temporal support now available in the DBMS.

## 2.4 Syntactically Similar Snapshot Reducibility

The syntactically similar snapshot reducibility (for short, S-reducibility) requirement aims at protecting the investments in programmer training and at ensuring continued efficient, cost-effective application development upon migration to a temporal model. This is achieved by exploiting the fact that programmers are likely to be comfortable with the non-temporal query language, e.g., SQL–92.

The requirement states that the query language of the temporally extended data model must offer, for each query in the non-temporal query language, a syntactically similar temporal query that is its "natural" generalization, in a precise technical sense. With this requirement satisfied and assuming that SQL–92 is the non-temporal query language, slightly modified versions of the SQL–92 queries, now on temporal tables, are temporal queries with semantics that are easily understood in terms of the semantics of the corresponding SQL–92 queries on snapshot tables. The familiar syntax and "naturally" extended semantics make it possible for programmers to immediately and easily write a wide range of temporal queries, with little need for expensive training, few errors, and no significant initial drop in productivity.

We first define the notion of snapshot reducibility among query languages. We will use $r$ and $r^{bi}$ for denoting a snapshot and a bitemporal relation instance, respectively. Similarly, $db$ and $db^{bi}$ are sets of snapshot and bitemporal relation instances, respectively. The bitemporal timeslice operator $\tau^{M^{bi},M}_{(c^{tt},c^{vt})}$ (e.g., [34, 7]) takes as arguments a bitemporal relation $r^{bi}$ (in the data model $M^{bi}$) and a bitemporal instant $(c^{tt}, c^{vt})$ and returns a snapshot relation $r$ (in the data model $M$) containing all tuples current at time $c^{tt}$ and valid at time $c^{vt}$. In other words, $r$ consists of all tuples of $r^{bi}$ whose associated time includes the time instant $(c^{tt}, c^{vt})$, but without the valid and transaction time.

**Definition 4 (snapshot reducibility)** [36] Let $M = (DS, QL)$ be a snapshot relational data model, and let $M^{bi} = (DS^{bi}, QL^{bi})$ be a bitemporal data model. Data model $M^{bi}$ is *snapshot reducible with respect to* data model $M$ iff

$$\forall q \in QL\ (\exists q^{bi} \in QL^{bi}\ (\forall db^{bi} \in DS^{bi}\ (\forall c^{tt}, c^{vt}\ (\tau^{M^{bi},M}_{(c^{tt},c^{vt})}(q^{bi}(db^{bi})) = q(\tau^{M^{bi},M}_{(c^{tt},c^{vt})}(db^{bi})))))). \quad \square$$

In other words, snapshot reducibility implies that for all query expressions $q$ in the snapshot model, there must exist a query $q^{bi}$ in the temporal model, such that for all $db^{bi}$ and for all time arguments, $q^{bi}$ reduces to $q$.

Observe that $q^{bi}$ being snapshot reducible with respect to $q$ poses no syntactical restrictions on $q^{bi}$. It is thus possible for $q^{bi}$ to be quite different from $q$, and $q^{bi}$ might be very involved even if $q$ s not. This is undesirable when we are formulating language design requirements and would like the temporal model to be a straightforward extension of the snapshot model. Consequently, we require that $q^{bi}$ and $q$ be syntactically similar.

**Definition 5 (syntactically similar snapshot reducibility)** [5] Let $M = (DS, QL)$ be a snapshot data model, and let $M^{bi} = (DS^{bi}, QL^{bi})$ be a bitemporal data model. Data model $M^{bi}$ is a *syntactically similar snapshot-reducible extension* of model $M$ iff

1. data model $M^{bi}$ is snapshot reducible with respect to data model $M$ and
2. there exist two (possibly empty) strings, $S_1$ and $S_2$, such that each query $q^{bi}$ in $QL^{bi}$ that is snapshot reducible with respect to a query $q$ in $QL$ is syntactically identical to $S_1 q S_2$.

If the two strings $S_1$ and $S_2$ are both the empty string, the extension is termed a syntactically identical snapshot reducible extension. □

The strings $S_1$ and $S_2$ are termed *statement modifiers* because they change the semantics of the entire statement $q$ that they enclose.

If the temporal data model treats temporal relations as new types of relations, it is possible to use the same syntactical constructs (i.e., $q^{bi}$ and $q$ are identical) for querying snapshot and temporal relations. In this case, the type of the argument relations determine the meaning of the construct. However, if TUC is also satisfied and if there is no separate, global context, it is impossible to achieve an extension that is *both* temporal upward compatible *and* syntactically identical snapshot-reducible.

With snapshot reducibility along with the syntactical similarity requirement satisfied, a snapshot reducible query evaluates to a result consistent with evaluating the syntactically similar, nontemporal query at each state of the argument temporal relation, producing a state of the output relation for each such evaluation. As a result, temporal queries are easily formulated and understood. This applies also to, e.g., modification statements and integrity constraints.

In the following examples, we prepend statements with the statement modifier `SEQ VT`, to be described in detail in Section 3. This modifier tells the temporal DBMS to evaluate statements with *sequenced* semantics in the valid-time dimension. We use the term "sequenced" to indicate that the database is viewed as a time-indexed sequence of snapshots. Explanations follow the example statements.

```
SEQ VT SELECT * FROM p;
SEQ VT SELECT p.A, q.C FROM p, q WHERE p.A = q.B;
SEQ VT SELECT A FROM p WHERE NOT EXISTS (SELECT * FROM q
                                        WHERE B > A);
SEQ VT SELECT AVG(B), C FROM q GROUP BY C;
CREATE TABLE r (D INTEGER, SEQ VT PRIMARY KEY (D));
```

The first query simply returns all tuples together with their valid time—this corresponds to returning the content of `p` at each state. The remaining queries assume that table `q` has also been altered to become a valid-time table. The second query joins `p` and `q` at each state of the database. This amounts to the well-known temporal natural join [19]. Similarly, the third query evaluates the query, and the subquery, on each state of the database, thereby performing a variation of temporal difference. Again, the modifier `SEQ VT` tells the DBMS to compute the difference at each snapshot. The last statement defines a table `r` and requires column `D` to be a "temporal" primary key, i.e., `D` must be a primary key at each state (but not necessarily across states).

**Example 2** Let `p` be $\{\langle 1\|5–8\rangle, \langle 3\|1–3\rangle, \langle 3\|4–12\rangle, \langle 4\|1–5\rangle\}$ and let `q` be $\{\langle 1, 2\|4–10\rangle, \langle 3, 2\|6-9\rangle, \langle 4, 2\|6-9\rangle\}$. Assume the queries shown above.

- The first query returns $\{\langle 1\|5-8\rangle, \langle 3\|1-3\rangle, \langle 3\|4-12\rangle, \langle 4\|1-5\rangle\}$.
- The second query returns $\{\langle 1, 2\|5-8\rangle, \langle 3, 2\|6-9\rangle\}$. Conceptually, we get the result by evaluation the enclosed SQL-statement on each state of the database. Computationally, the interval $6-9$ is the result of intersecting the intervals $4-12$ and $6-9$ (interval intersection returns those instants that are contained in both input intervals).
- The third query returns $\{\langle 1\|5-5\rangle, \langle 3\|1-3\rangle, \langle 3\|4-5\rangle, \langle 3\|10-12\rangle, \langle 4\|1-5\rangle\}$. Again, we conceptually evaluate the enclosed statement on each state of the database. Computationally, we, e.g., subtract the interval 6–9 from the interval $4-12$ to get the intervals $4-5$ and $10-12$.
- The aggregate query returns $\{\langle 1, 2\|4-5\rangle, \langle 2.667, 2\|6-9\rangle, \langle 1, 2\|10-10\rangle\}$. As before, we evaluate the enclosed statement on each state to determine the result. □

**Example 3** Let `r` be $\{\langle 1\|3-6\rangle, \langle 1\|10-17\rangle, \langle 2\|4-8\rangle\}$. Assume the data definition statement shown above.

- The database is consistent because at each state `r.D` is a primary key.
- Adding $\langle 1\|7-9\rangle$ to `r` leaves the database in a consistent state.
- Adding $\langle 2\|7–9\rangle$ to `r` violates the consistency because `r.D` would then not be a primary key at times 7 and 8. □

These examples illustrate that syntactically similar snapshot reducible statements are easy to write and understand. However, despite their natural semantics, these statements become very difficult to write without statement modifiers. For example, even skilled SQL programmers will find it close to impossible to formulate all examples in pure SQL.

## 2.5 Interval Preservation

While the novel coupling of the well-known snapshot reducibility property (Definition 4) with syntactical similarity (Definition 5) and the use of this resulting property as a guideline for how to syntactically embed temporal functionality in a language is highly attractive, it is also limited. Specifically, S-reducibility does not distinguish between different relations if they contain the same snapshots, i.e., if they are snapshot equivalent [19]. This means that many different results of an S-reducible query are generally possible: the results will be snapshot equivalent, but differ in how the result tuples are timestamped. As a simple example, if $\{\langle X \| 1-5\rangle\}$ is a possible result of an S-reducible query, so is $\{\langle X \| 1-2\rangle, \langle X \| 3-5\rangle\}$. This section delves into the issue of which result, or results, should be favored out of the many possible results permitted by S-reducibility. To illustrate the issue in more detail, consider Figure 1.

*president*

| Name | VTIME |
|---|---|
| Bill Clinton | 1993/01/20 – 1997/01/20 |
| Bill Clinton | 1997/01/21 – $NOW$ |

*president′*

| Name | VTIME |
|---|---|
| Bill Clinton | 1993/01/20 – $NOW$ |

Figure 1: Snapshot-equivalent Relations with Different Application Semantics

The two relations in Figure 1 are different and may be given different meanings by the user. The tuples in the relation to the left denote the fact "Bill Clinton was president" during two adjacent time periods. The relation to the right represents when Bill Clinton was president, which is different from recording during which terms he served. In spite of this difference, the relations are the same in the eyes of snapshot reducibility. This is so because to snapshot reducibility, a relation is no more than a sequence of snapshot relations; and the two relations are mutually snapshot equivalent, i.e., for all time points $tp$, the snapshots $\tau_{tp}^{vt}(president)$ and $\tau_{tp}^{vt}(president')$ are identical. We thus have an example where two different relations—with quite different meanings in terms of what is important for the application user—cannot be told apart by snapshot-equivalence-based properties.

The difference between the two relations is that one is coalesced while the other is not [9]. In general, two tuples in a temporal relation with intervals as timestamps are candidates for coalescing if they have identical explicit attribute values and adjacent or overlapping timestamps. Such tuples may arise in many ways. For example, uncoalesced tuples may have been stored in the database on purpose, update operations may not enforce coalescing due to efficiency concerns, or a projection of a coalesced temporal relation may produce an uncoalesced result, much as duplicate tuples may be produced by a projection on a duplicate-free snapshot relation.

When formulating more specific design requirements for how to timestamp tuples of query results, two possibilities come to mind. We can require results to be coalesced. This solution is attractive because it defines a canonical representation for temporal relations. Potential disadvantages are that timestamps of tuples stored into the database are are not preserved and that with more than one interval-valued time dimension, no unique coalesced relation exists (cf. Section 3.2). As the second possibility, we can preserve, or respect, the timestamps as originally entered into the database. This approach is faithful to the information entered by the user and offers more control to the user, but it also moves the responsibility for maintaining the semantics of the timestamps from the system to the user.

Because there are advantages to both possibilities, we have chosen to illustrate how statement modifiers may accomodate both in the same language. The default is to preserve the timestamps—being irreversible, coalescing cannot be the default.

In the sequel we define interval preservation [2], which intuitively requires that timestamps are respected as much as possible, i.e., unless required by the snapshot reducibility requirement, timestamps may not be split or merged. As first steps, we introduce three auxiliary notions, namely *normalized relations, relevant tuples,* and *maximal interval fragments.*

To define normalized relations, we use the following auxiliary notions from the literature. Relations $R_1$ and $R_2$ are *snapshot equivalent,* i.e., $R_1 \overset{se}{=} R_2$, iff at each point in time their snapshots are identical [19]. Two tuples are *value equivalent* iff their explicit attributes are pairwise identical [9]. Finally, a *temporal element* is a finite union of intervals, i.e., a set of maximal non-overlapping intervals [14].

We use the database and query from Example 2 for illustrating the definitions that follow. Thus, define:

- $\texttt{p} = \{\langle 1 \| 5-8\rangle, \langle 3 \| 1-3\rangle, \langle 3 \| 4-12\rangle, \langle 4 \| 1-5\rangle\}$
- $\texttt{q} = \{\langle 1, 2 \| 4-10\rangle, \langle 3, 2 \| 6-9\rangle, \langle 4, 2 \| 6-9\rangle\}$
- $\texttt{DB} = \texttt{p} \cup \texttt{q}$
- $\texttt{Q} =$ `SEQ VT SELECT A FROM p`
  `WHERE NOT EXISTS (SELECT * FROM q WHERE B > A)`
- $\texttt{r} = \{\langle 1 \| 5-5\rangle, \langle 3 \| 1-3\rangle, \langle 3 \| 4-5\rangle, \langle 3 \| 10-12\rangle, \langle 4 \| 1-5\rangle\}$

Normalized relations are timestamped with temporal elements and do not contain value-equivalent tuples. A set of snapshot equivalent relations shares the same normalized relation, and we use normalized relations to characterize the set of time points that must be included in the timestamp of a query result. Note that because all intervals in normalized relations are maximal, the actual result intervals are guaranteed to be subintervals of the intervals in a normalized relation.

**Definition 6 (normalized relation)** Let $R$ be a temporal relation and let $R^n$ be a temporal relation timestamped with temporal elements. $R^n$ is *normalized* with respect to $R$ iff $R^n$ is snapshot equivalent with $R$ and $R^n$ does not contain value-equivalent tuples. □

**Example 4** Relations p, q, and r from above are normalized as follows.

- $\texttt{p}^n = \{\langle 1 \| \{5-8\}\rangle, \langle 3 \| \{1-12\}\rangle, \langle 4 \| \{1-5\}\rangle\}$
- $\texttt{q}^n = \{\langle 1, 2 \| \{4-10\}\rangle, \langle 3, 2 \| \{6-9\}\rangle, \langle 4, 2 \| \{6-9\}\rangle\}$
- $\texttt{r}^n = \{\langle 1 \| 5-5\rangle, \langle 3 \| \{1-5, 10-12\}\rangle, \langle 4 \| \{1-5\}\rangle\}$. □

We proceed to define the notion of relevant tuples. To each tuple of a normalized result relation, there exists a set of tuples that is relevant to it, i.e., if any of the relevant tuples is removed from the input database, the query no longer yields a relation that is snapshot equivalent with the normalized result relation. Relevant tuples are interesting because they contribute to the computation of the result. The timestamps of relevant tuples are the ones that need to be preserved. Note that the set of relevant tuples is generally not homogeneous, but may include tuples from relations with different schemas.

**Definition 7 (relevant tuples)** Assume a temporal database $DB$ and a S-reducible query $Q$. $DB^{rel}_{t^n} \subseteq DB$ is the *relevant tuples* for $t^n \in Q(DB)^n$ iff $Q(DB^{rel}_{t^n}) \overset{se}{=} \{t^n\}$ and $\forall DB'(DB' \subset DB^{rel}_{t^n} \Rightarrow \neg(Q(DB') \overset{se}{=} \{t^n\}))$. □

**Example 5** With `DB`, `Q`, and `r` from our example the following holds true.

- The relevant tuples for $\mathtt{t}^n_1 = \langle 3 \| \{1–5, 10–12\} \rangle$ are $\mathtt{DB}^{rel}_{t^n_1} = \{\langle 3 \| 1–3 \rangle, \langle 3 \| 4–12 \rangle, \langle 4, 2 \| 6-9 \rangle\}$.
- The relevant tuple for $\mathtt{t}^n_2 = \langle 4 \| \{1-5\} \rangle$ is $\mathtt{DB}^{rel}_{t^n_2} = \{\langle 4 \| 1-5 \rangle\}$. □

Intuitively, the maximal interval fragments for a normalized tuple of a query result are those parts of the intervals of the relevant tuples that must be included in the query result. Maximal fragments are important because we want to preserve intervals as much as possible in query results (but without violating snapshot reducibility). We use $time(t)$ to denote the timestamp of tuple $t$, and we let intersections ($\cap$) return temporal elements.

**Definition 8 (maximal interval fragments)** Assume a temporal database $DB$ and an S-reducible query $Q$ on $DB$. Let $t^n \in Q(DB)^n$, the normalized query result, and let $DB^{rel}_{t^n}$ be the set of relevant tuples for $t^n$. The *maximal interval fragments (MIF)* are then given as follows.

$$MIF(DB, Q, t^n) = \{I | \exists t_i \in DB^{rel}_{t^n} \wedge I \in (time(t_i) \cap time(t^n))\} \qquad \Box$$

The maximal interval fragments are thus the intervals obtained from intersecting the timestamps of relevant tuples with the timestamp of the normalized tuple.

**Example 6** Assume $\mathtt{t}^n_1$, $\mathtt{t}^n_2$, $\mathtt{DB}^{rel}_{t^n_1}$, and $\mathtt{DB}^{rel}_{t^n_2}$ from the previous example.

- The maximal interval fragments for $\mathtt{t}^n_1$ and $\mathtt{DB}^{rel}_{t^n_1}$ are $\{1-3, 4-5, 10-12\}$.
- The maximal interval fragment for $\mathtt{t}^n_2$, and $\mathtt{DB}^{rel}_{t^n_2}$ is $\{1-5\}$. □

We are now in a position to define interval preservation. Informally, interval preservation means that the timestamp of a result tuple is restricted to an interval from the corresponding set of maximal interval fragments.

**Definition 9 (interval preservation)** An S-reducible query $Q$ is *interval preserving* iff for all databases $DB$, each tuple in $Q(DB)$ is timestamped with an interval from the set of maximal interval fragments of the corresponding normalized tuple. □

**Example 7** In our example, we have $\texttt{r} = \{\langle 1\|5-5\rangle, \langle 3\|1-3\rangle, \langle 3\|4-5\rangle, \langle 3\|10-12\rangle, \langle 4\|1-5\rangle\}$. The maximal interval fragment for the tuple with explicit attribute value 1 is $\{5-5\}$. The maximal interval fragments for tuples with explicit attribute value 3 are $\{1-3, 4-5, 10-12\}$. The maximal interval fragment for the tuple with explicit attribute value 4 is $\{1-5\}$. Thus, intervals are preserved and the query is interval preserving.

On the other hand, assume $\texttt{r}_1 = \{\langle 1\|5-5\rangle, \langle 3\|1-5\rangle, \langle 3\|10-12\rangle, \langle 4\|1-5\rangle\}$, which results from coalescing the two value-equivalent tuples in $\texttt{r}$ with adjacent intervals. Because $\texttt{r}_1 \overset{se}{=} \texttt{r}$, the result is perfectly acceptable according to the snapshot reducibility. However, intervals are not preserved because $1-5$ is not a maximal interval fragment.

Finally, assume $\texttt{r}_2 = \{\langle 1\|5–5\rangle, \langle 3\|1–3\rangle, \langle 3\|4–5\rangle, \langle 3\|10–12\rangle, \langle 4\|1–2\rangle, \langle 4\|3–5\rangle\}$, which we obtain from splitting an interval in $\texttt{r}$. Again, $\texttt{r}_2 \overset{se}{=} \texttt{r}$, which means that $\texttt{r}_2$ respects the snapshot reducibility requirement. Yet, intervals are not preserved because neither $1-2$ nor $3-5$ are elements of the corresponding set of maximal interval fragments. □

## 2.6 Universal Statement Modifiers

The reducibility requirement of Definition 5 is applicable only to queries of the underlying non-temporal query language. Thus, it does not extend to queries such as the following.

```
SEQ VT
  SELECT *
  FROM p, q
  WHERE p.X = q.X
  AND DURATION(VTIME(p),YEAR) > 5
```

```
SEQ VT
  SELECT p.X, VTIME(q),
              VTIME(p)
  FROM p, q
  WHERE p.X = q.X
```

Both queries are quite natural and easy to understand. The query to the left constrains the temporal join to p-tuples with a valid time longer than 5 years. Note that this condition cannot be evaluated on individual snapshots because the timestamp is lost when taking a snapshot of a temporal database. The query to the right computes a temporal join as well, but also returns the original valid times. Again snapshot reducibility by itself cannot be used to answer the query because the original valid times are not present in the snapshots.

Queries such as these arise naturally. DBMSs generally provide predicates and functions on time attributes, which may be applied to, e.g., valid time. Enlarging the applicability of the SEQ VT modifer to statements that include predicates and functions on valid and transaction time yields a more user-friendly (and orthogonal) query language.

**Definition 10 (universal statement modifiers)** Statement modifiers are *universal* iff they apply to all statements. □

Although S-reducibility in itself cannot define statements that include functions and predicates on timestamps, these statements are constrained to observe the spirit of S-reducibility (they are consistent with viewing a temporal database as a time indexed sequence of nontemporal databases). Recall the sample query above. The condition `DURATION(VTIME(p),YEAR) > 5` cannot be evaluated by considering individual snapshots in isolation. However, the temporal join itself can still be conceptualized as a nontemporal join evaluated on each snapshot. Thus, S-reducibility can be used to constrain the semantics of the "nontemporal constructs" (e.g., a join, difference, or subquery) of a sequenced statement, but it cannot be used to define temporal constructs that explicitly reference the timestamps.

Statements that may include functions and predicates on timestamps are defined in Section 4. Specifically, the statements are mapped into well–defined temporal relational algebra expressions. The temporal algebra is carefully designed to (1) respect snapshot reducibility, (2) define the timestamps of query results, and (3) preserve timestamps. A formal discussion and proofs follow later in the paper.

## 2.7 Non-Sequenced Statements

Sequenced statements are attractive because they provide built-in temporal semantics that is based on viewing a database as a sequence of states. However, there are many reasonable queries that cannot be expressed as sequenced queries. Therefore a temporal query language should also allow *non-sequenced* queries, with no built-in temporal semantics enforced.

**Definition 11 (non–restrictiveness)** A query language is *non–restrictive* iff timestamps can be manipulated like regular attributes, with no implicit temporal semantics enforced. □

Non-restrictiveness is attractive because it guarantees a standard non-temporal behavior of statements. This is particularly important in the context of a smooth migration where users can be expected to be well-acquainted with the semantics of their non-temporal language. The non-restrictiveness requirement ensures that users are able to keep using the paradigm they are familiar with and to incrementally adopt the new features.

There is a theoretical argument in favor of non-restrictive languages as well: It has been shown that any variant of temporal logic, a well-developed language that only provides built-in temporal semantics, is strictly less expressive than a first order logic language with explicit references to time, i.e., a non-restrictive language [42].

We use the modifier `NSEQ VT` to signal non-sequenced semantics, i.e., standard semantics with full explicit control over timestamps. (The choice of this modifier is discussed in the next Section.)

```
NSEQ VT
   SELECT *
   FROM p, q
   WHERE VTIME(p) PRECEDES VTIME(q) AND A = B;
CREATE TABLE s (E INTEGER, NSEQ VT PRIMARY KEY (E));
```

The query joins `p` and `q`. The join is not performed at each snapshot. Instead we require that the valid time of `p` precedes the valid time of `q`. The result relation is a nontemporal one. The data definition statement makes `E` a nonsequenced primary key of `r`, i.e., independent of the time, `E` is a primary key of `r`.

**Example 8** Let `p` be $\{\langle 1\|5-8\rangle, \langle 3\|1-12\rangle, \langle 4\|1-5\rangle\}$ and let `q` be $\{\langle 1,2\|4-10\rangle, \langle 3,2\|6-9\rangle, \langle 4,2\|6-9\rangle\}$. The above query returns $\{\langle 4\rangle\}$. □

**Example 9** Let `s` be $\{\langle 1\|3-6\rangle, \langle 2\|4-8\rangle\}$.

- The database is consistent because, independent of the time, `s.E` is a primary key.
- Adding another tuple with an explicit attribute value of either 1 or 2 violates the consistency of the database because this makes `s.E` no longer a primary key that is independent of the time.
- Adding a tuple with an explicit attribute value other than 1 and other than 2 leaves the database in a consistent state. □

The concept of non-sequenced queries naturally generalizes to modifications. Non-sequenced modifications destructively change states, with information retrieved from possibly all states of the original relation.

Non-sequenced statements are more complex than S-reducible statements in the sense that the user gets less built-in support. The query language must provide a set of functions and predicates so that the user can express temporal relationships (e.g., `PRECEDES`) and perform manipulations and computations on timestamps (e.g., `VTIME`). This requires new constructs in the query language. These constructs are, however, easy to integrate because they require changes at the level of built-in predicates and functions only.

## 2.8 Summary of Requirements

In this section, we have formulated requirements that are essential for the data model of a temporal DBMS to satisfy in order to ensure a smooth transition from a non-temporal DBMS to the temporal system and in order to ensure a systematic and comprehensive support for advanced temporal statements. We review each in turn.

Upward compatibility guarantees that replacing the existing DBMS with a new, temporal DBMS does not affect the functioning of any application code. Temporal upward compatibility guarantees that, with the new DBMS in place, it is possible to incrementally exploit more and more of the built-in temporal support. Specifically, changing existing snapshot relations to become temporal relations does not affect the functioning of any legacy code. Together, these two requirements aim at making it possible to benefit from temporal support while protecting the investments in legacy application code.

The requirement that the temporal language be a syntactically similar snapshot-reducible extension of the existing language makes the temporal query language easy to use for programmers familiar with the existing query language. They ensure a systematic support for temporal statements that conceptualize a temporal database as a sequence of nontemporal database states. For such queries, the temporal system automatically computes the timestamps of the result queries. This is very attractive because the alternative is to explicitly formulate the predicates necessary to correctly compute the timestamps, which is often a complicated and error-prone activity, leading to involved and hard-to-understand statements. We have shown simple statements that would become extremely difficult to formulate without the availability of built-in temporal support.

Another important observation is that snapshot reducibility cannot be used to define the timestamps to be returned. We therefore require a language to be interval preserving. This not only defines the timestamps to be returned but it also ensures that the database system is faithful to the timestamps stored in the database. In order to increase the utility of statement modifiers and in order to increase the orthogonality of the language, we require that statement modifiers be universally applicable, i.e., modifiers can be applied to statements that include predicates and functions on timestamps.

Finally, when the semantics of sequenced queries are not adequate, the language should provide good support for expressing the intended timestamps and results within nonsequenced statements. For example, a set of predicates on timestamps should be available that allow for the convenient expression of the possible ordering relations among timestamps. The non-restrictiveness requirement gives the user full control over timestamps.

## 3 Applying Statement Modifiers to SQL-based Languages

The previous section motivated and defined requirements without making restrictive assumptions about the properties of particular query languages and data models—simple SQL-based examples were used merely for illustration purposes.

The main purpose of this section is to describe the language design space constrained by the requirements and to demonstrate the practical utility of statement modifiers for meeting the requirements. To achieve this, we have chosen to develop a design of an SQL-based temporal language. We have chosen SQL–92 as the concrete context because it is a rather complex language and because of its widespread use. However, statement modifiers are not restricted to a specific language, but are generally applicable.

## 3.1 Global Impact of Requirements

Upward compatibility dictates that the temporal language contains all statements of SQL–92, including its temporal features. For example, SQL–92 contains the data type `INTERVAL` of duration values. Thus, a new language should also use `INTERVAL` for durations, and another keyword must be chosen for the interval data type—we choose `PERIOD`. As another implication, the temporal extension must contend with *all* the facilities of SQL–92, e.g., nested queries, aggregates, and null values.

In order to satisfy temporal upward compatibility (TUC), it is necessary that all SQL–92 statements work on temporal relations as well as on snapshot relations, as described in detail in the previous section. This is achieved by letting SQL–92 modification statements on temporal relations modify the current and future states of the relations. The statements thus take effect on the states current at the time of the modification, and the effects persist in the (dynamically changing) current state from that time on and until affected by other modifications. Queries, views, and constraints simply consider only the snapshot states of the argument temporal relations that are current and valid at the times they are evaluated. This semantics guarantees that adding time to existing snapshot relations has no effect on the applications that use them.

The requirement that there should exist syntactically similar snapshot reducible temporal counterparts of all SQL–92 queries also affects the design. For each SQL–92 query, we must be able to pre- or append a fixed text string, a *modifier*, to get the corresponding temporal query. We chose `SEQ VT` for valid time, `SEQ TT` for transaction time to emphasize that the temporal database is viewed as a sequence of nontemporal databases. Note that it is not possible to use the empty modifier, which is reserved for TUC statements.

Sequenced statements offer built-in, or default, timestamp-related processing—the temporal DBMS rather than the application does the potentially very complex processing involving timestamps. Thus, it is an attractive property of the new query language that as many queries as possible can be formulated as sequenced queries. This reduces the complexity of application code, with many associated benefits. To increase the utility of sequenced queries, we extend them with so-called domain specifications, making it possible to restrict the parts of the argument tuples considered in queries to certain time periods. We also add range specifications that allow the specification of the timestamps of result tuples. These specifications are integral parts of the statement modifiers.

While the built-in semantics of sequenced queries are "natural" in the specific technical sense defined earlier, there are many queries that cannot be formulated using these default semantics. Rather, it must be possible to formulate a much wider range of queries where the application programmer is in complete control of, and responsible for, the timestamp manipulation. Such queries need another modifier, different from that of sequenced queries. Using no flags is not an option, due to interaction with the semantics of SQL–92 queries on temporal relations, as dictated by temporal upward compatibility. We choose the flags `NSEQ VT` and `NSEQ TT`. In these nonsequenced queries, no default timestamp-related semantics is built into the query language. Rather, the timestamps of temporal relations are made available in the query, essentially as regular, explicit attributes.

The new period data type, `PERIOD`, is used for the timestamps. In addition, built-in facilities for constructing periods and for end-point extraction are provided along with a host of predicates on the data type (cf. Appendix A).

## 3.2 Adding Detail to the Design

The requirements shape the overall design of a temporal extension of SQL as discussed above. When we move to a more detailed level in the design, good design practice (e.g., generality and orthogonality) rather than the requirements guides the design. Below, we add detail to the general design (Section 4 provides precise semantics).

### Extensions at the Statement Level

First, we discuss how to associate modifiers with statements, i.e., with query expressions, views, assertions, integrity constraints, and modification statements.

We saw that different semantics are given to different temporal statements: (i) SQL–92 statements, (ii) statements with semantics dictated by temporal upward compatibility, (iii) sequenced statements, and (iv) statements with nonsequenced semantics. Now, we study in more detail the syntax of the statement modifiers that specify these semantics.

Section 2 simply requires that a statement modifier is placed at the beginning or end of a statement and that it applies to the statement as a whole. Within these restrictions, there are several possibilities for the positioning of the statement modifiers for the different types of statements. We provide an EBNF syntax for each of our specific extensions to SQL–92. We thus focus on the temporal extensions and gloss over the details of SQL–92. In addition, we will focus on queries, which are the most complex statements, but will also consider other statements. In the EBNF productions that follow, terminals are of the form "xxx", i.e., enclosed in quotation marks. Non-terminals of the form <xxx> derive from the SQL–92 standard [23, p.481ff], and new non-terminals are of the form <xxx>. Omitting these new non-terminals yields the original (slightly simplified) SQL–92 productions.

- In *queries* and cursor expressions (termed <cursor specification> in SQL–92) the statement modifiers are placed at the outermost level, right at the beginning.

  ```
  <cursor specification> ::=
     <modifiers> <query expression>
     [ <order by clause> ] |
     "(" <modifiers> <query expression> ")" <coal>
     [ <order by clause> ]
  ```

  The scope of the semantics implied by the statement modifiers is all parts of the query (e.g., including nested queries), with the exception of derived table expressions in the from clause. The non-terminal <coal> is used for specifying coalescing, to be discussed later in this section.

- In *views*, the statement modifiers are placed immediately following the AS keyword.

  ```
  <view definition> ::=
     "CREATE" "VIEW" <table_name>
     [ "(" <view column list> ")" ] "AS"
     ( <modifiers> <query expression> |
      "(" <modifiers> <query expression> ")" <coal> )
  ```

- Statement modifiers can be associated with *derived table expressions* in from clauses. The motivation is that derived tables may be meaningfully computed independently of the computation of the remainder of the containing query. Put differently, derived table expressions have their own scope and may be replaced by views or auxiliary tables, thus meaningfully allowing derived tables expressions to have their own individual statement modifiers. This adds flexibility to the language and improves its usefulness. As a syntactic shorthand, coalescing is also allowed after table names in the from clause (in order to facilitate point-based queries).

  ```
  <table reference> ::=
     <table name> <coal>
     [ [ "AS" ] <correlation name> ] |
     "(" <modifiers> <query expression> ")" <coal>
     [ "AS" ] <correlation name>
  ```

  Note that, while syntactically similar, derived tables in the from clause are quite different from subqueries in the where clause. Subqueries can be correlated with the main query and cannot be evaluated independently. Therefore, no separate modifier is allowed for subqueries.

- In *assertions*, statement modifiers are placed right after the CHECK keyword.

```
<assertion definition> ::=
   "CREATE" "ASSERTION" <constraint name>
   CHECK <modifiers> "(" <search condition> ")"
```

- *Table and column constraints* are syntactic shorthands for assertions. The statement modifiers are placed right in front of the table and column constraints, respectively.

```
<column definition> ::=
   <column name> <modifiers>
   <column constraint definition>

<table constraint definition> ::=
   <constraint name definition> <modifiers>
   <table constraint>
```

- As with queries, the modifiers are placed in front of *modification statements*.

```
<SQL data change statement> ::=
   <modifiers> <insert statement> |
   <modifiers> <delete statement> |
   <modifiers> <update statement>
```

Summarizing, statement modifiers are associated with all “statements” that can be evaluated independently. Examples include queries, data manipulation statements, assertions, integrity constraints, and views. In general, statement modifiers are placed in front of statements to emphasize their impact upon the entire statement.

### Statement Modifiers

We start with an EBNF syntax for statement modifiers and continue with a discussion of their meaning.

```
<modifiers> ::= [ <modifier> [ "AND" <modifier> ] ]
                [ <vt_range> ]
<modifier>  ::= <mode> <dimension> [ <domain> ]
<mode>      ::= "SEQ" | "NSEQ"
<dimension> ::= "TT" | "VT"
<domain>    ::= period_constant
<vt_range>  ::= "SET" "VT" period_expression
```

The meaning of the statement modifiers naturally divides into four orthogonal parts, namely the specification of the core semantics, the time-domain specification, the time-range specification, and specification of coalescing. We start with the discussion of the core semantics and continue with domain and range specifications and coalescing in the next sections.

The following three types of modifiers determine the *core semantics* of temporal statements. Each of the three types of modifiers applies orthogonally to valid and transaction time.

<***empty modifier***> A missing modifier for a time dimension (i.e., valid or transaction) dictates upward compatibility (UC) when neither of the underlying argument relations support that time; otherwise, evaluation according to temporal upward compatibility (TUC) is dictated. For queries, the time dimension will not be present in the result relation.

`SEQ` When this keyword is present for a time dimension, evaluation consistent with sequenced semantics (SEQ), i.e., built-in timestamp-related processing, is dictated for the time dimension. The time dimension will be present in relations that result from queries.

`NSEQ` This keyword signals nonsequenced semantics (NSEQ), i.e., timestamp processing with no built-in semantics enforced by the temporal DBMS. The affected time dimension is not present in query results (with this modifier, the time effectively becomes an explicit attribute that can be included in the result similarly to how other explicit attributes are included).

With two time dimensions, the three cases lead to a total of nine kinds of statements, as summarized in Table 1. For simplicity, we have omitted permutations of the valid and transaction time modifier (they have the same semantics).

| syntax | semantics | |
|---|---|---|
| | vt | tt |
| <SQL–92> | (T)UC | (T)UC |
| `SEQ VT` <SQL–92> | SEQ | (T)UC |
| `NSEQ VT` <SQL–92> | NSEQ | (T)UC |
| `SEQ TT` <SQL–92> | (T)UC | SEQ |
| `NSEQ TT` <SQL–92> | (T)UC | NSEQ |
| `SEQ VT AND SEQ TT` <SQL–92> | SEQ | SEQ |
| `SEQ VT AND NSEQ TT` <SQL–92> | SEQ | NSEQ |
| `NSEQ VT AND SEQ TT` <SQL–92> | NSEQ | SEQ |
| `NSEQ VT AND NSEQ TT` <SQL–92> | NSEQ | NSEQ |

Table 1: The Basic Usage of Statement Modifiers

### Time-Domain and Time-Range Specifications

Statement modifiers also allow for time-domain and time-range specifications. The *time domain* is a period constant that may be placed right after the VT and TT keywords, respectively. It restricts the database to the part that is valid or current during the respective period. A domain restriction is applied prior to the evaluation of a statement, i.e., in a preprocessing step.

```
SEQ VT PERIOD ′1994-1997′
  SELECT * FROM p, q WHERE p.A = q.B;
CREATE TABLE r (C INTEGER,
                SEQ VT PERIOD ′10-20′ PRIMARY KEY (C));
```

The domain restriction in the query says that we are only interested in facts valid during the last four years. Similarly, it is possible to restrict integrity constraints to a certain period. Specifically, the primary key constraint will only be enforced from time 10 to time 20.

For valid time, it can be meaningful to specify the valid time of the result, i.e., the *time range*. The SET VT clause is used for this purpose. Note that it makes no sense to provide a similar clause for transaction time. Transaction-time semantics forbids this kind of user interaction [28]. The time range is set in a postprocessing step, i.e., after the evaluation of a query.

```
NSEQ VT
SET VT PERIOD(BEGIN(VTIME(p)),END(VTIME(q)))
  SELECT * FROM p, q WHERE VTIME(p) PRECEDES VTIME(q);
```

The statement joins p- and q-tuples if the former precedes the latter. The valid time of the result tuple is set to the period that covers the valid times of both input tuples including all time points in-between.

### Coalescing

Coalescing merges tuples with overlapping or adjacent timestamps, and identical corresponding attribute values (termed value equivalent), into a single tuple. Coalescing is allowed at the levels where the modifiers are also allowed. In addition, as a syntactic shorthand, a coalescing operation is permitted directly after a relation name in the from clause. In this case, a coalesced instance of the relation, rather than the uncoalesced one, is considered.

```
  <coal> ::=  "(" <dimension> ")"
```

The semantics of coalescing depends on the type of relation it is applied to. A snapshot relation cannot be coalesced. A valid-time relation can be coalesced in valid time only, and the equivalent is true for transaction-time relations. With a single time dimension, coalescing degenerates to the merging of value-equivalent tuples with overlapping or adjacent time periods. In this case, the meaning is straightforward (performance aspects of one-dimensional coalescing have been studied elsewhere [9]).

We thus turn our attention to the coalescing of bitemporal relations where the semantics are more subtle. Here, overlapping or adjacent time regions (rectangles) of value-equivalent tuples have to be merged. In the general case, overlapping rectangles do not coalesce into a single rectangle, which means that several result tuples have to be generated. This can be done in two ways: with the resulting rectangles maximized in valid time or in transaction time. We use `(VT)` for the former and `(TT)` for the latter. Figure 2 exemplifies bitemporal coalescing.

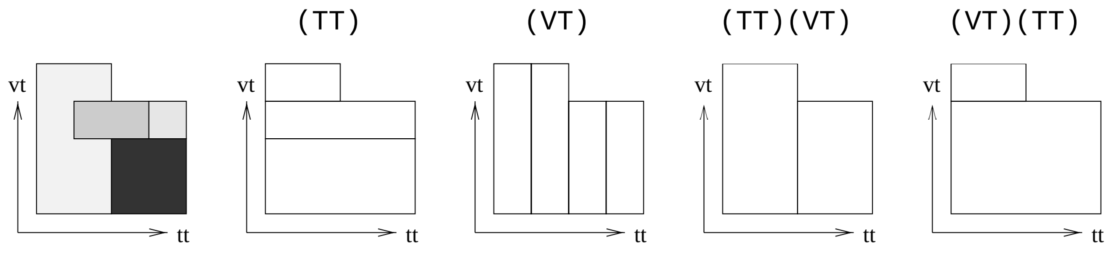


Figure 2: Different Forms of Coalescing

The first picture displays the rectangular shapes defined by the timestamps of four value-equivalent tuples. The second and third pictures illustrate the two basic coalescing operations; coalescing in transaction time and coalescing in valid time. These two basic operations can be combined to `(TT)(VT)`, which means that we first coalesce in transaction time and then in valid time. As exemplified by the last two pictures, the sequence of coalescing operations matters. Sequence `(TT)(VT)` results in maximal valid-time periods, whereas `(VT)(TT)` results in maximal transaction-time periods.

```
(SEQ VT SELECT * FROM p)(VT);
SEQ VT SELECT * FROM p(VT) WHERE DURATION(VTIME(p),YEAR) > 5;
```

In the first statement, we coalesce the result of a sequenced query. In the second query, we coalesce the relation in the from clause because we want the subsequent condition to be evaluated over maximal valid times only.

## 3.3 Summary of Syntax

Initially, we discussed the global impact of the requirements. We then defined the syntax of a temporal extension of SQL–92 that satisfies UC, TUC, SEQ, and NSEQ in terms of EBNF productions. We emphasized the general concepts with the goal of making it easier to appreciate that it is possible to extend a non-temporal language different from SQL–92, as well as to use different specific modifiers.

## 4 A Formal Semantics

This section provides a precise and comprehensive definition of the semantics of the temporal extension of SQL in terms of a mapping to standard and temporal relational algebra, both of which are defined here. We illustrate that statement modifiers are amenable to giving a concise and precise definition of the semantics.

### 4.1 Translating Temporal Statements to Relational Algebra Expressions

The translation to (temporal) relational algebra expressions consists of two parts. First, we consider constructs at the level of functions and predicates. This step is straightforward and is discussed in the first section. The translation at the statement level, i.e., the translation of statements enhanced with statement modifiers, is much more involved (and important!) and is covered in the subsequent three sections.

#### Constructs for Timestamp Manipulation

Temporal query languages generally define a variety of constructs to manipulate their various timestamp types. These include constructors (to create instances of the timestamp types), extractors (to extract constituent parts from timestamps), predicates (boolean-valued, for comparison), and operations (to create new timestamps from existing ones). Many constructs exist in the literature [39, pp. 251–291]. They are relatively easy to define, and adding one more construct to a language has only a localized effect on the language design. Therefore, we only define a relatively small number of constructs here.

We assume the most common timestamp representation, namely four `TIMESTAMP` attributes representing valid and transaction time, respectively. This representation leads to the definitions given in Appendix A, which we will use throughout, including in relational algebra expressions, e.g., in selection predicates. This makes the expressions more readable. It is straightforward to adapt these definitions to different representations, e.g., a representation that is based on the `PERIOD` data type of the evolving SQL/Temporal part of the SQL3 standard.

#### Query Expressions

We define the meaning of temporal query expressions by translating them to well-defined algebraic expressions. As a precursor, we introduce the notation that we will use in the algebra expressions.

We use $\langle t \rangle$, $\langle t \| VT \rangle$, $\langle t \| TT \rangle$, and $\langle t \| VT, TT \rangle$ to denote tuple variables ranging over snapshot, valid-time, transaction-time, and bitemporal relations, respectively. The vertical double-bar "$\|$" is used to separate the explicit attributes from the implicit timestamps. The valid time is referred to as $VT$, the transaction time as $TT$.

In the definitions, we need auxiliary operators that timeslice relations and turn timestamps into regular, explicit attributes. These operators are overloaded to apply to valid-time, transaction-time, and bitemporal relations, and they have variants for both valid and transaction time. Their formal definition is provided in Appendix B. There are two timeslice operations. The first, $\tau_{tp}$, selects all tuples in the argument relation with a timestamp that overlaps time point $tp$. The time dimension used in this selection is not present in the result relation. The second timeslice operation, $\delta_{per}$, returns all argument tuples that overlap with period $per$. The timestamp of a result tuple is the intersection of $per$ with the tuple's original timestamp. The snapshot operation $SN$ turns a time dimension into an explicit attribute. Note that $SN$ is not needed at the implementation level, where all attributes are explicit (cf. Section 4.1). With these conventions in place, Table 2 gives the semantics for core statements (cf. Table 1).

In the table, $[\![\text{<SQL–92>}]\!]_{SQL-92}$ evaluates to the standard relational algebra expression that corresponds to <SQL–92> [12, 18]. Next, $[\![\text{<SQL–92>}]\!]_T$, where $T \in \{vt, tt, bi\}$, evaluates to the same algebraic expression as does $[\![\text{<SQL–92>}]\!]_{SQL-92}$, except that every nontemporal relational algebra operator (e.g., $\times, \sigma, \pi$) is replaced by the corresponding temporal relational algebra operator (e.g., $\times^T, \sigma^T, \pi^T$). The algebras are defined in Section 4.2. The following two examples illustrate the definition.

**Example 10** The query below, termed $Q_1$, is an example of a non-sequenced query. The argument relations are assumed to be bitemporal.

```
NSEQ VT
   SELECT p.X
   FROM p, q
   WHERE p.X = q.X
   AND VTIME(p) PRECEDES VTIME(q)
```

This query is defined by the relational algebra expression given next.

$$[\![Q_1]\!]_{temp}(p, q) = \pi_{p.X}(\sigma_{p.X=q.X}(\sigma_{VTIME(p)\ PRECEDES\ VTIME(q)}(SN^{vt}(\tau^{tt}_{now}(p)) \times SN^{vt}(\tau^{tt}_{now}(q)))))$$

Note that the mapping from SQL–92 queries to relational algebra is still the same. The temporal selection condition can be viewed as a syntactic shorthand for a standard selection condition (cf. Table 6). The only addition is the "adjustment" of the relations ($SN^{vt}$ and $\tau^{tt}_{now}$) to fit the non-sequenced evaluation mode for valid time and the temporal upward compatible evaluation mode for transaction time. □

$$[\![\text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{tt}_{now}(\tau^{vt}_{now}(r_1)), \ldots, \tau^{tt}_{now}(\tau^{vt}_{now}(r_n)))$$

$$[\![\texttt{SEQ VT}\ \text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{vt}(\tau^{tt}_{now}(r_1), \ldots, \tau^{tt}_{now}(r_n))$$

$$[\![\texttt{NSEQ VT}\ \text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{tt}_{now}(\mathit{partSize2e}^{vt}(r_1)), \ldots, \tau^{tt}_{now}(\mathit{partSize2e}^{vt}(r_n)))$$

$$[\![\texttt{SEQ TT}\ \text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{tt}(\tau^{vt}_{now}(r_1), \ldots, \tau^{vt}_{now}(r_n))$$

$$[\![\texttt{NSEQ TT}\ \text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{SQL-92}(\tau^{vt}_{now}(\mathit{partSize2e}^{tt}(r_1)), \ldots, \tau^{vt}_{now}(\mathit{partSize2e}^{tt}(r_n)))$$

$$[\![\texttt{SEQ VT AND SEQ TT}\ \text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{bi}(r_1, \ldots, r_n)$$

$$[\![\texttt{SEQ VT AND NSEQ TT}\ \text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{vt}(\mathit{partSize2e}^{tt}(r_1), \ldots, \mathit{partSize2e}^{tt}(r_n))$$

$$[\![\texttt{NSEQ VT AND SEQ TT}\ \text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{tt}(\mathit{partSize2e}^{vt}(r_1), \ldots, \mathit{partSize2e}^{vt}(r_n))$$

$$[\![\texttt{NSEQ VT AND NSEQ TT}\ \text{<SQL–92>}]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$
$$[\![\text{<SQL–92>}]\!]_{SQL-92}(\mathit{partSize2e}^{tt}(\mathit{partSize2e}^{vt}(r_1)), \ldots, \mathit{partSize2e}^{tt}(\mathit{partSize2e}^{vt}(r_n)))$$

Table 2: Core Semantics

**Example 11** The following query, termed $Q_2$, is sequenced in both valid and transaction time.

```
SEQ VT AND SEQ TT
   SELECT p.X
   FROM p, q
   WHERE p.X = q.X
```

It is defined by the following temporal relational algebra expression.

$$[\![Q_2]\!]_{temp}(p, q) = \pi^{bi}_{p.X}(\sigma^{bi}_{p.X=q.X}(p \times^{bi} q))$$

Apart from the superscripts on the operators, the translation from SQL–92 queries to relational algebra expressions remain the standard one. □

## Domain and Range Specifications

Next, we define the semantics of domain and range specifications. A time-domain restriction constrains the argument relations in a query to contain only tuples that are valid during a specific period. Thus, only the parts of argument tuples that intersect with the time-domain restriction are considered when the query is evaluated. This is formalized in Table 3.

$$\begin{aligned}
&[\![\text{<mode>}\ \mathsf{VT}\ \text{<domain>}\ q^T]\!]_{temp}(r_1, \ldots, r_n) \overset{\triangle}{=} \\
&\qquad [\![\text{<mode>}\ \mathsf{VT}\ q^T]\!]_{temp}(\delta^{vt}_{\text{<domain>}}(r_1), \ldots, \delta^{vt}_{\text{<domain>}}(r_n)) \\
&[\![\text{<mode>}\ \mathsf{TT}\ \text{<domain>}\ q^T]\!]_{temp}(r_1, \ldots, r_n) \overset{\triangle}{=} \\
&\qquad [\![\text{<mode>}\ \mathsf{TT}\ q^T]\!]_{temp}(\delta^{tt}_{\text{<domain>}}(r_1), \ldots, \delta^{tt}_{\text{<domain>}}(r_n))
\end{aligned}$$

Table 3: Definition of Domain Restrictions

Next, it is possible to specify time ranges—using the modifier "SET VT *range*" where *range* is period valued—that determine the valid times of the result tuples. There are two different situations. First, if the core statement is a SEQ VT statement then the automatically computed valid time is replaced by the value resulting from evaluating the time-range specification. Second, for all other core statements, prepending SET VT *range* results in the inclusion of valid time into the result. Because these core statements return results that do not contain valid-time timestamps, the type of the result is changed. The valid time of a tuple is that resulting from evaluating *range*. The details are given in Table 4.

$$[\![\mathsf{SET\ VT}\ \mathit{range}\ q^T]\!]_{temp}(r_1, \ldots, r_n) \overset{\triangle}{=}$$

$$\begin{cases}
\{\langle t \| VT\rangle \mid \langle t\rangle \in [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \wedge VT = \mathit{range}(t)\} \\
\qquad \text{if } [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \text{ evaluates to a snapshot relation} \\
\{\langle t \| VT\rangle \mid \langle t \| VT'\rangle \in [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \wedge VT = \mathit{range}(t)\} \\
\qquad \text{if } [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \text{ evaluates to a valid-time relation} \\
\{\langle t \| VT, TT\rangle \mid \langle t \| TT\rangle \in [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \wedge VT = \mathit{range}(t)\} \\
\qquad \text{if } [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \text{ evaluates to a transaction-time relation} \\
\{\langle t \| VT, TT\rangle \mid \langle t \| VT', TT\rangle \in [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \wedge VT = \mathit{range}(t)\} \\
\qquad \text{if } [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \text{ evaluates to a bitemporal relation}
\end{cases}$$

Table 4: Definition of Range Specifications

## Coalescing

Any query that returns a temporal relation may be coalesced. To define coalescing, let $q^T$ denote any temporal query. If this query returns a valid-time relation, it may be modified to $(q^T)(\mathsf{VT})$, to return the coalesced version of the valid-time relation. The obvious corresponding result holds when replacing valid time by transaction time. If the query returns a bitemporal relation, it may be coalesced in valid time, in transaction time, or in a combination of the two. Table 5 provides the definitions. Definitions of representative versions of the algebraic coalescing operator, *coal*, will be given shortly.

$$[\![(q^T)(\mathsf{VT})]\!]_{temp}(r_1, \ldots, r_n) \overset{\triangle}{=}$$
$$\begin{cases} coal^{vt}([\![q^T]\!]_{temp}(r_1, \ldots, r_n)) \\ \qquad \text{if } [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \text{ evaluates to a valid-time relation} \\ coal^{bi}_{vt}([\![q^T]\!]_{temp}(r_1, \ldots, r_n)) \\ \qquad \text{if } [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \text{ evaluates to a bitemporal relation} \end{cases}$$
$$[\![(q^T)(\mathsf{TT})]\!]_{temp}(r_1, \ldots, r_n) \overset{\triangle}{=}$$
$$\begin{cases} coal^{tt}([\![q^T]\!]_{temp}(r_1, \ldots, r_n)) \\ \qquad \text{if } [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \text{ evaluates to a transaction-time relation} \\ coal^{bi}_{tt}([\![q^T]\!]_{temp}(r_1, \ldots, r_n)) \\ \qquad \text{if } [\![q^T]\!]_{temp}(r_1, \ldots, r_n) \text{ evaluates to a bitemporal relation} \end{cases}$$

Table 5: Definition of Coalescing

## 4.2 The Temporal Relational Algebra

Having provided mappings from the temporal extension of SQL to a combination of conventional and temporal relational algebra expressions, the next step is to define the algebra operators that may occur in these expressions.

We start by reviewing Codd's relational algebra. In the definitions given in Figure 3, $c$ is a predicate and $f$ is a generalized projection function that roughly corresponds to the select list of an SQL–92 statement.

We proceed by defining the temporal relational algebra operators. With the exception of the Cartesian product, the operators respect snapshot reducibility (Section 5 studies this in detail). In addition, two other properties of the algebra are noteworthy. First, the algebra is interval-based in that it preserves the timestamps in the relations. It thus generally matters for query results whether, e.g., one tuple

$$
\begin{array}{lcl}
\sigma_c(r) & \triangleq & \{t \mid t \in r \wedge c(t)\} \\
\pi_f(r) & \triangleq & \{t_1 \mid \exists t_2(t_2 \in r \wedge t_1 = f(t_2))\} \\
r_1 \cup r_2 & \triangleq & \{t \mid t \in r_1 \vee t \in r_2\} \\
r_1 \times r_2 & \triangleq & \{t_1 \circ t_2 \mid t_1 \in r_1 \wedge t_2 \in r_2\} \\
r_1 \setminus r_2 & \triangleq & \{t \mid t \in r_1 \wedge t \notin r_2\}
\end{array}
$$

Figure 3: The Snapshot Relational Algebra

with valid time $10-20$ or two (value-equivalent) tuples with valid times $10-15$ and $16-20$, appear in an argument relation. Second, care was taken to only consider end points of valid and transaction timestamps when defining the operators—intermediate time points are never used. This allows for an efficient (essentially, granularity independent) implementation.

Figure 4 contains the definition of the valid-time version of the temporal relational algebra. The transaction-time version is omitted because it is similar, the only difference being that the temporal operations are performed on the transaction-time attribute rather than on the valid-time attribute. The definition uses function *intersect* (on two periods) and the predicate *overlaps* (on two periods), both of which were defined in Table 6. The symbol "$\circ$" denotes tuple concatenation.

$$
\begin{array}{lcl}
\sigma_c^{vt}(r) & \triangleq & \{\langle t \| VT\rangle \mid \langle t \| VT\rangle \in r \wedge c(\langle t, VT\rangle)\} \\
\pi_f^{vt}(r) & \triangleq & \{\langle t_1 \| VT\rangle \mid \exists t_2(\langle t_2 \| VT\rangle \in r \wedge t_1 = f(\langle t_2, VT\rangle))\} \\
r_1 \cup^{vt} r_2 & \triangleq & \{\langle t \| VT\rangle \mid \langle t \| VT\rangle \in r_1 \vee \langle t \| VT\rangle \in r_2\} \\
r_1 \times^{vt} r_2 & \triangleq & \{\langle\langle t_1, VT_1\rangle \circ \langle t_2, VT_2\rangle \| VT\rangle \mid \langle t_1 \| VT_1\rangle \in r_1 \wedge \langle t_2 \| VT_2\rangle \in r_2 \wedge \\
& & \quad VT = intersect(VT_1, VT_2) \wedge \\
& & \quad VT_1\, overlaps\, VT_2\} \\
r_1 \setminus^{vt} r_2 & \triangleq & \{\langle t \| VT\rangle \mid \exists VT_1(\langle t \| VT_1\rangle \in r_1 \wedge \\
& & \quad (\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \\
& & \quad \vee VT^- = VT_1^-) \wedge \\
& & \quad (\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \\
& & \quad \vee VT^+ = VT_1^+) \ \wedge VT^- < VT^+ \wedge \\
& & \quad \neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \ \wedge VT_4\, overlaps\, VT))\}
\end{array}
$$

Figure 4: The Valid-Time Algebra

Clearly the most complex operation is temporal difference. In the general case, three tuples are required to determine one result tuple, namely one tuple from $r_1$ and two tuples from $r_2$, as illustrated in Figure 5.

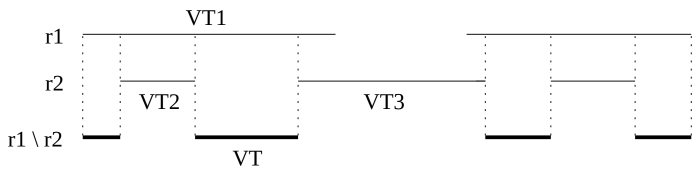


Figure 5: Valid-Time Difference

The second line of the definition in Figure 4 identifies all potential starting points for periods of result tuples. Result periods may start where a period from an $r_1$ tuple starts and where a period of a $r_2$ tuple ends. The second line then identifies all potential end points of periods of result tuples. The last two lines of the definition then exclude "false" result tuples: The third line eliminates meaningless combinations of starting and ending points, and the last line eliminates tuples with excessive periods.

The only operation without a non-temporal counterpart is coalescing. It is also special because it destroys the representation of timestamps in order to enforce a particular representation (maximum periods). By definition, coalescing merges (chains of) overlapping or adjacent value-equivalent tuples as illustrated in Figure 6.

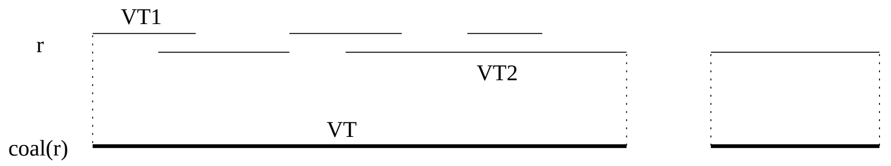


Figure 6: Coalescing a Valid-Time Relation

While it is not possible to compute arbitrary transitive closures in SQL–92, coalescing is possible in SQL–92 because time is *linear* [9, 11].

$$
\begin{aligned}
coal^{vt}(r) \;\triangleq\; \{\langle t \| VT\rangle \mid\ & \exists VT_1\, \exists VT_2(\langle t\|VT_1\rangle \in r \wedge \langle t\|VT_2\rangle \in r \wedge \\
& VT_1^- < VT_2^+ \wedge VT^- = VT_1^- \wedge VT^+ = VT_2^+ \wedge \\
& \forall VT_3(\langle t\|VT_3\rangle \in r \ \wedge VT^- < VT_3^- < VT^+ \Rightarrow \\
& \quad \exists VT_4(\langle t\|VT_4\rangle \in r \wedge VT_4^- < VT_3^- \leq VT_4^+)) \wedge \\
& \neg\exists VT_5(\langle t\|VT_5\rangle \in r \wedge \\
& \quad (VT_5^- < VT^- \leq VT_5^+ \vee VT_5^- \leq VT^+ < VT_5^+)))\}
\end{aligned}
$$

The two tuples introduced in the first line serve to define the starting ($VT_1^-$) and end ($VT_2^+$) points of a coalesced tuple, as specified in the second line. The third and fourth lines ensure that there are no gaps between $VT^-$ and $VT^+$. This is done by ensuring that every tuple with a start time between $VT^-$ and $VT^+$ is extended towards $VT^-$, i.e., there must exist another tuple with a valid time containing the respective start time. Finally, on the last line we make sure that the valid time of the result tuple is maximal, i.e., there may not exist another tuple that contains either $VT^-$ or $VT^+$.

The bitemporal relational algebra is a natural extensions of the valid-time (and transaction-time) algebra. However, both time dimensions must be handled simultaneously, meaning that rectangles rather than periods must be considered. While this does not change the basic ideas, it adds to the complexity of the definitions; for this reason, it is deferred to Appendix C.

### 4.3 Summary of Semantics

We defined the semantics of temporal queries in three steps. First, the semantics of constructs for timestamp manipulation was given. The second step was to define the semantics of core queries, as well as queries with domain and range specifications and coalescing. Specifically, mappings to relational and temporal relational algebraic expressions were given. Finally, the relational and temporal relational algebras were defined.

Taking into consideration that a temporal extension of SQL–92 is a language more complicated than SQL–92, the semantics are quite concise. This has been achieved by giving the semantics in terms of the semantics of SQL–92 (specifically, in terms of a mapping of SQL–92 queries to relational algebra). This, in turn, is possible because statement modifiers systematically and faithfully extend SQL–92.

## 5 Properties of the Temporal Statement Modifier-Extended SQL

This section discusses properties of temporal statement modifieres. It is argued that the extension indeed satisfies the compatibility and reducibility properties that were introduced in Section 2. We also contrast temporal statement modifiers with approaches based on (syntactic) defaults. For brevity and to avoid tedious details, we cover the valid-time dimension only.

### 5.1 Compatibilities and Syntactic Restrictions

Upward compatibility with respect to SQL–92 (or any other language to be extended) is fulfilled by design. The approach adopted for defining the syntax and semantics emphasizes this property: The syntax was given by *extending* the syntax of SQL–92 with *non-mandatory* constructs. Thus, the new language contains all legal SQL–92 statements. The approach taken to define the semantics also makes it straightforward to verify that all SQL–92 statements retain their original semantics. Specifically, the first definition in Table 2 covers SQL–92 statements.

The statement modifiers of Section 3 also ensure temporal upward compatibility with SQL–92. This follows from the satisfaction of upward compatibility, the first definition in Table 2, and the definition of SQL–92 modification statements when applied to temporal relations. These statements are covered in detail elsewhere [3].

Finally, snapshot reducible statements (further discussed below) are syntactically similar with respect to SQL–92. This again follows from the definition of the language. Definition 5 constrains the differences between an SQL query and the corresponding syntactically similar snapshot reducible temporal query to be at most two fixed strings (i.e., the statement modifiers), prepended and appended to the SQL query, respectively. The fixed strings do not depend on the particular query, but are the same for all queries. Satisfying this requirements leads to a wholesale, “semantic” approach to defaults, as will be discussed in Section 5.3.

Statement modifiers are universal because they apply to all statements without restriction. Nonsequenced statements fulfill the non-restrictiveness requirement because they allow manipulation of timestamps as ordinary explicit attributes, with no implicit temporal semantics being enforced (cf. the last definition in Table 2).

## 5.2 Syntactically Similar Snapshot Reducibility

In this section the focus of attention is the S-reducibility property of sequenced queries with respect to SQL–92. The satisfaction of this property follows from the design of the mapping and the definition of the temporal algebraic operators. Below, we discuss first how the definition of the temporal algebra is shaped to make the preservation of the property possible. Then follows a discussion of the top-level mapping of sequenced statements (as given in Table 2).

Recall the definition of snapshot reducibility (Definition 4). We first show that the valid-time relational algebra almost has this property with respect to the snapshot relational algebra.

**Theorem 1** The valid-time relational algebra (Figure 4) satisfies the following reducibility properties below with respect to the snapshot relational algebra (Figure 3).

- $\forall tp\ (\tau_{tp}^{vt}(\sigma_c^{vt}(r)) \equiv \sigma_c(\tau_{tp}^{vt}(r)))$
- $\forall tp\ (\tau_{tp}^{vt}(\pi_f^{vt}(r)) \equiv \pi_f(\tau_{tp}^{vt}(r)))$
- $\forall tp\ (\tau_{tp}^{vt}(r_1 \cup^{vt} r_2) \equiv \tau_{tp}^{vt}(r_1) \cup \tau_{tp}^{vt}(r_2))$
- $\forall tp\ (\pi_{r_1.VT, r_2.VT}^{-}(\tau_{tp}^{vt}(r_1 \times^{vt} r_2)) \equiv \tau_{tp}^{vt}(r_1) \times \tau_{tp}^{vt}(r_2))$
- $\forall tp\ (\tau_{tp}^{vt}(r_1 \setminus^{vt} r_2) \equiv \tau_{tp}^{vt}(r_1) \setminus \tau_{tp}^{vt}(r_2))$

Recall that *tp*, *c*, and *f* denote a time point, a predicate, and a projection list, respectively. The equivalences hold for arbitrary relations, with the only restrictions being that in the first two equivalences, $c$ and $f$ refer to explicit attributes only and that the relations be union compatible in the third and fifth equivalence. Also, $\pi^-_X(r)$ is given by $\pi_{r.*\backslash X}(r)$ where $r.*$ denotes all the attributes of $r$. □

The proofs of these properties may be found in Appendix D.

It follows that the valid-time selection, projection, union, and difference operators are snapshot reducible to their snapshot counterparts. Thus, all valid-time algebra statements involving only these operators are snapshot reducible to the snapshot algebra statements obtained by simply removing the *vt* superscripts.

However, the equivalence involving the Cartesian products attracts attention: this operator is not reducible to the snapshot Cartesian product! While it is straightforward to define a temporal Cartesian product that is snapshot reducible to the snapshot Cartesian product, we have chosen a definition that violates snapshot reducibility. Let us explore why this is a good design decision.

Initially, note that the “problem” with our temporal Cartesian product is that it retains the implicit valid-time attributes of its argument relations and turns them into explicit attributes. The operator $\pi^-$ is introduced to eliminate these “extraneous” attributes. Now, when mapping a (sequenced) temporal query to its algebraic equivalent, we would like to exploit the standard mapping used when mapping SQL queries to relational algebra. Consider the following query.

```
SEQ VT
    SELECT <L>
    FROM p, q, r
    WHERE <P>
```

We would like to map this query to

$$\pi^{vt}_{<L'>}(\sigma^{vt}_{<P'>}((p \times^{vt} q) \times^{vt} r))$$

where $\langle L' \rangle$ and $\langle P' \rangle$ are slight syntactical variations of `<L>` and `<P>`, respectively. One possible choice for predicate `<P>` would be

```
        DURATION(VTIME(p),DAY) + DURATION(VTIME(q),DAY) <
        DURATION(VTIME(r),DAY)
```

With our definition of the valid-time Cartesian product, we can express the corresponding algebra predicate $\langle P' \rangle$ as follows because the timestamps of the argument tuples are retained as explicit attributes.

$$\mathit{DURATION}(p.VT, DAY) + \mathit{DURATION}(q.VT, DAY) < \mathit{DURATION}(r.VT, DAY)$$

Using a snapshot-reducible Cartesian product would make it impossible to construct a corresponding predicate $\langle P' \rangle$. The information required to evaluate the predicate would be lost. This observation holds for any tuple timestamped and any homogeneous [15] attribute-value timestamped data model. Snapshot-reducible temporal Cartesian products for such models are unable to serve the role during the mapping of temporal SQL queries to algebraic expressions that the snapshot Cartesian product serves when mapping SQL queries to relational algebra.

One approach to retain the simple mapping and also retain a snapshot reducible temporal Cartesian product is to introduce an additional (information-preserving) Cartesian product that produces results with *two implicit* valid times. But this latter product returns results that are not valid-time relations and thus breaks the closedness property of the algebra, an undesirable complication. This approach was adopted in the algebra for the HSQL data model [30] that includes both a reducible “Concurrent Product” and an information-preserving “Cartesian product.”

Another approach that will ensure that the necessary information is available in the algebra for evaluating any predicate `P` is to introduce an n-ary valid-time join that can then be defined to be snapshot reducible. The transformation to algebra would then be as follows.

$$\pi^{vt}_{\langle L'\rangle}(\bowtie^{vt}_{\langle P'\rangle}(p, q, r))$$

This approach was adopted for the algebra proposed for TSQL2 [35]. While the added complexity of an n-ary operator may be undesirable, there is another problem with this approach. Consider the sample `<L>` = `p.X, VTIME(p)` that specifies that the implicit valid-time attribute of relation `p` is to be present in the result as an explicit attribute. With the n-ary join approach, it is not possible to produce an equivalent $\langle L'\rangle$. Specifically, the original valid times of tuples also from `p` cannot be inferred from the result of the join. With our Cartesian product, we have $\langle L'\rangle = p.X, p.VT$.

Most temporal algebras have operators that are snapshot reducible with respect to the snapshot Cartesian product (e.g., the TJOIN [25], the Concurrent Product Operator [30], the cross-product [27], (temporal) equijoin [10], and the valid-time Cartesian product[3] [38]; reference [22] gives a survey).

The simple binary temporal Cartesian product defined here permits the use of the standard mapping from SQL to algebra without imposing any restrictions on the contents of the `SELECT` and `WHERE` clauses. As we discuss next, the non-reducibility of the operator does not lead to violations of the S-reducibility to SQL.

In SQL-based languages, Cartesian products are specified using the `FROM` clause of *query specifications* [23, p.175]. For a temporal query to be reducible, the result of evaluating it must not include the implicit valid-time attributes of argument tuples as explicit attributes. In S-reducible queries, it is not possible to select a time

[3]This operator is defined in a non-homogeneous attribute-value timestamped data model. Unlike any other product we have seen, this operator reduces to the snapshot Cartesian product and yet does not possess the two deficiencies.

dimension of a relation; and defaults (e.g., `SELECT *`) do not expand to include the implicit time attributes. The presence of subsequent projections in the definition of reducible queries, the presence of the additional explicit time attributes in the results of Cartesian products then does not compromise the S-reducibility property.

The choice of making the temporal Cartesian products in isolation not snapshot reducible improves the design of the temporal SQL, improving its orthogonality and ease of use. Another choice would have been to have the user turn valid times into explicit attributes (using derived table expressions) and to make Cartesian products snapshot reducible, but this is less attractive because some statements become cumbersome to formulate.

## 5.3 Built-in Semantics and Defaults

The additional user-friendliness achieved when extending a query language with temporal support has two sources. First, the addition of *new temporal data types* with associated constructors, predicates, and operations makes temporal data management more convenient. For example, adding a period data type to SQL–92 makes it easier to manage the valid time of tuples. To illustrate this, assume that snapshot relation `p` has attributes `a` and `VT`, with the latter being period-valued and recording valid time. Similarly, let relation `q` have attributes `b`, `c`, and `VT`. A temporal natural join of these two relations is expressed as follows.

```
SELECT p.a, q.b, q.c, INTERSECT(p.VT, q.VT) AS VT
FROM p, q
WHERE p.a = q.b AND p.VT OVERLAPS q.VT
```

Without a data type for time periods, using instead pairs of time-point valued attributes, this query gets harder to formulate and understand.

Second, providing *built-in timestamp processing* adds user-friendliness. For example, with statement modifiers it is possible to write a temporal natural join essentially as a regular natural join, as follows.

```
SEQ VT
  SELECT p.a, q.b, q.c
  FROM p, q
  WHERE p.a = q.b
```

In this query, we assume that `p` and `q` are valid-time relations. The statement modifier triggers a sequenced evaluation, which means that the subsequent snapshot natural join turns temporal.

For a simple join query, the added user-friendliness of using built-in processing over simply using a new data type is clear, but not substantial. This is so because it is relatively straightforward to generalize a snapshot natural join to a temporal natural join. When considering complicated SQL–92 queries, possibly involving subqueries and aggregates, the generalized queries frequently become very difficult to formulate in SQL–92, with or without new data types. This is where the utility of built-in processing stands out. For example, with statement modifiers *any* query is generalized by simply prepending `SEQ VT`.

Two approaches to built-in processing may be identified. The first approach is represented by TSQL2 [39, e.g., pp. 291–297], which provides comprehensive built-in processing. In this approach, built-in processing is provided by *syntactically defined defaults*. For example, in TSQL2 the above join can be formulated as follows.

```
SELECT p.a, q.b, q.c                  SELECT p.a, q.b, q.c
FROM p, q                             VALID INTERSECT(p, q)
WHERE p.a = q.b                       FROM p, q
                                      WHERE p.a = q.b
```

The query to the left is a syntactic shorthand for the query to the right. When the valid clause of TSQL2 is missing from a query and all argument relations are valid-time relations, the meaning of the query is given by adding a valid clause that generates a timestamp of result tuples that is given by the intersection of the timestamps of the argument tuples.

The second approach is the *"semantic" approach* we have adopted for statement modifiers. Rather than defining query language statements that provide built-in processing in terms of syntactical additions to them, the built-in processing is defined semantically, i.e., to be consistent with S-reducibility.

The syntactic approach has been shown to be problematic. We have previously identified TSQL2 statements that have no obvious semantics [5]. The problem stems in part from the syntactic defaults. For example, the rule above becomes unclear when select statements are included in the where clause (subqueries). It turns out that it is exceedingly difficult to define built-in processing in SQL–92 via syntactic defaults in a manner that is comprehensive and also systematic and thus sufficiently easily comprehensible for it to be practically useful.

The problem with syntactic defaults is one of scalability over language constructs, and SQL–92 has numerous constructs with subtle semantics and also lacks orthogonality. With syntactic defaults, it must be possible to state in the query language the default for a wide range of statements. Accomplishing this is quite challenging because each syntactic default is dependent on the specifics of the statement that it is the default for. In this way, the complexity of specifying the actual default is comparable to the complexity of specifying the entire language.

Semantic defaults behave quite differently. No attempt is made to actually define defaults that are syntactical shorthands for other, more cumbersome statements. Specifically, with statement modifiers we do not try to syntactically map sequenced statements to semantically equivalent (non-sequenced) statements. (Our experience with compiling sequenced statements to SQL suggests that this is impractical.) This makes the temporal language conceptually simple, and it becomes much more robust with respect to extensions and dialects of SQL because the SQL part of a temporal statement can essentially be treated as a black box.

In fact, the temporal statement modifier-extended SQL may be seen as the result of replacing in TSQL2 the syntactic defaults by systematic, semantically-based built-in time-related processing, thereby fixing fundamental problems in TSQL2 [5]. The semantic approach leads to a syntactically identifiable class of queries with built-in support and thus provides a systematic and wholesale approach to built-in default processing.

### 5.4 Summary of Properties

The section covered first the properties that the temporal SQL was designed to fulfill, namely UC, TUC, S-reducibility, universal statement modifiers, and non-restrictiveness. It then proved the S-reducibility of sequenced queries. Finally, semantic and syntactic defaults were compared.

## 6 Related Work

We cover in turn related work on language requirements and related temporal SQL's.

### 6.1 Query Language Requirements

We describe the background of the language requirements from Section 2, as well as related requirements.

Few precise query language requirements have been proposed by other authors. While the phrase "upward compatibility" has been used widely and in many contexts [1, p. 513], [26, p. 99], [30, p. 123], [30, p. 125], [21, p. 480], we have found no precise definition of it.

The precise formulation of the UC and TUC requirements in Section 2 evolved in part from studies of TSQL2 [39] and were developed with Richard Snodgrass and John Bair [3]. Extensive discussions in the context of developing proposals for the SQL/Temporal part [32, 31] of SQL3 shaped the formulation of the requirements.

We have encountered one proposal that aims at satisfying a requirement that seems similar to temporal upward compatibility. The TempSQL language (e.g., [17]) introduces notions of so-called classical and system user types. System users see the full temporal database, while classical users see only the current snapshot of the database. If applications are classical by default, and if individual statements, rather than all statements issued by a user, can be independently made temporal, this would essentially (providing that a number of other design decisions are made correctly) yield a temporal upward compatible SQL extension.

The formulation of the notion of S-reducibility uses the fundamental notion of snapshot reducibility [36] and was inspired by an informal concept that was presented in the context of the ChronoLog language. S-reducibility was formalized during the process of solving identified problems in TSQL2 [5], the goal being to develop a proposal for standardization in SQL3 [32, 31, 33].

The language requirements beyond S-reducibility, i.e., interval preservation, universality of statement modifiers, and non-restrictiveness, do not appear to have been subject to study by other authors. Theoretical aspects of interval preservation are the subject of an upcoming paper [2]. We are not aware of requirements that relate to universal statement modifiers and non-restrictive languages.

## 6.2 Temporal Extensions to SQL

We first consider related efforts that involve statement modifiers, then consider each existing temporal SQL in turn.

### Statement Modifiers

Temporal statement modifiers represent a novel approach for adding temporal support to an existing language. A primitive form of statement modifiers were used in ChronoLog [8], a temporal extension of a Datalog-based language.

Earlier offsprings of the work described in this paper are the change proposals that were submitted to the SQL/Temporal part of SQL3 [32, 31, 33] (also cited above). The change proposals evolved through extensive interactions with the ANSI and ISO standardization committees. This interaction led to syntactical and semantic changes aimed at making the proposals palatable to context of the the standard, with its associated peculiarities. The extension presented here differs from that of the change proposals in several respects. It satisfies a comprehensive set of requirements; emphasizes orthogonality of language constructs; has a formal, deterministic semantics. The present paper also discusses and proves properties of statement modifiers.

### Existing Temporal SQL's

To complete the coverage of related research, we evaluate all existing temporal SQL proposals that we are aware of, including SQL–92, with respect to our requirements, and and we relate their designs to the statement modifier-based approach. We report compliance with a requirement if this is claimed in the documentation of a proposal, or if non-compliance cannot be proved. We consider each SQL in turn and in chronological order of their appearance. UC is satisfied by all proposals, and a more detailed study of the TUC requirement may be found elsewhere [3].

TOSQL [1] extends SQL with the specification of the query's time aspects. These extensions include AT, WHILE, DURING, BEFORE, AFTER, ALONG, and

AS_OF clauses. The default options are defined syntactically such that a query that omits the temporal portion retains the standard meaning of the corresponding SQL select operation. TUC, while not defined explicitly, was clearly a concern when designing TOSQL. A large part of TOSQL respects TUC, but statements of the form `select * from r` violate TUC because they also return the table's timestamp(s). S-reducibility is not provided. The built-in time-related processing is restricted to the above clauses and can be overwritten by stating the clauses explicitly, which makes TOSQL non-restrictive and interval preserving.

TSQL [24, 25, 26] is a superset of SQL, extending the latter with, e.g., a WHEN, a TIME-SLICE, and a MOVING WINDOW clause. TSQL satisfies neither TUC (no adequate defaults for the above mentioned clauses) nor S-reducibility. The restriction to coalesced relation instances also breaks the interval preservation requirement. Apart from the enforced coalescing, TSQL is non-restrictive.

HSQL [29, 30] is again a superset of SQL. The retrieval facilities are enhanced with facilities for coalescing (COALESCES, COALESCE ON), concurrent products, timeslicing (FROMTIME t1 TOTIME t2), and unfolding (EXPAND). The concurrent product provides built-in snapshot reducible semantics for joins (and products), but not for, e.g., subqueries, aggregates, set difference, and integrity constraints. Like TOSQL, the design of HSQL takes TUC into consideration. For the same reasons as TOSQL, it does not achieve complete satisfaction. HSQL is interval preserving and non-restrictive. It does not support S-reducibility.

SQL–92 [23] provides only quite limited support for temporal data. SQL–92 is not temporal upward compatible with itself (legacy statements are not restricted to the current time). The S-reducibility property is not satisfied (no built-in processing of, e.g., temporal joins). Because SQL–92 does not treat time with special semantics, it is trivially interval preserving and non-restrictive.

TempSQL [4, 16, 17] is an extension of SQL defined over relations where attribute values are temporal assignments, i.e., partial functions from time into the domain of the attribute. Temporal expressions $[\![\ldots]\!]$, which extract the time domain of attribute values or relations, is a prominent feature of TemdSQL. Temporal expressions can be used in (nested) expressions. The SELECT-FROM-WHERE statement is extended with a WHILE clause that may be used for specifying the time domain of a tuple [17, p. 39]. As discussed previously, TUC is only satisfied for so-called classical users that see only the current state of all relations. When a classical-user needs access to past states of a relation and is made a so-called system user, the full application must be rewritten, breaking TUC. S-reducibility is not provided. TempSQL is restrictive in the sense that set operations have an enforced, built-in temporal semantics. TempSQL provides automatic coalescing, violating interval preservation.

IXSQL [21] provides support for generic interval data in SQL. It extends SQL–92 with a REFORMAT AS and a NORMALIZE ON clause. The reformat clause is used to specify a sequence of UNFOLD and FOLD operations, which convert a set of intervals into a set of constituent points, and vice versa. The NORMALIZE operation is a syntactic abbreviation of the reformat clause, and it coalesces a relation. For the same reason as for SQL–92, it does not satisfy TUC. No support for S-reducibility is provided. IXSQL is non-restrictive and to some degree, it is also interval preserving (if UNFOLD is used, intervals are not preserved because interval boundaries are lost when unfolding a table).

ChronoSQL [8] was designed to illustrate how to carry over the predecessor of our statement modifiers from a deductive to an SQL-based language. ChronoSQL includes a REDUCE construct, with which it is possible to achieve S-reducibility. TUC is not achieved. ChronoSQL is interval preserving, non-restrictive, and the REDUCE is generally applicable.

As discussed earlier, TSQL2 [39] employs syntactic defaults. It adds a VALID clause to SQL–92 for specifying the timestamp of the result. If the VALID clause is omitted from a query, intersection semantics is assumed. By default TSQL2 returns valid time relations. To retrieve a snapshot relation, SELECT SNAPSHOT has to be specified. TSQL2 neither satisfies TUC (valid-time relations are returned by default) nor S-reducibility (subqueries and relations with duplicates violate this [5]). TSQL2 is not interval preserving because coalesced instances are enforced. While some operations come with a hard-wired implicit temporal semantics (e.g., set operations applied to temporal relations), TSQL2 is non-restrictive in the sense that implicit timestamps can be rendered explicit.

The change proposals submitted to the SQL standardization committee [32, 31, 33] describe early work on temporal statement modifiers (cf. above). They were designed to fulfill TUC and S-reducibility. Interval preservation is not guaranteed because timestamps of snapshot reducible queries are left unspecified in the non-deterministic definition of the language. Non-restrictiveness is achieved via non-sequenced statements. Statement modifiers are not universal because sequenced modifiers may only be applied to legacy, i.e., nontemporal, statements.

## 7 Summary and Research Directions

The paper discusses how to use temporal statement modifiers to manage temporal information. It takes as its outset a number of syntactic and semantic requirements, motivated by real-world concerns, that a temporal data model and query language must satisfy to contend with legacy applications, permit the coexistence of non-temporal and temporal data, and exploit the programmers' expertise with the non-temporal language of their choice. Care was taken to make the requirements independent of any particular data model, although we explore them in the relevant and challenging context of SQL. No existing model or language satisfies all of these requirements. In particular, this paper is the first one to formulate a comprehensive set of requirements that combines salient features of temporal languages (snapshot reducibility, temporal upward compatibility) with salient features of nontemporal languages (interval preservation, non-restrictiveness).

The next step was to explore and exemplify how these requirements shape a temporal extension of SQL. A statement modifier-based extension makes it possible to adopt a black-box approach to defining the new language, leading to a concise definition of a comprehensive temporal query language that covers core as well as advanced language features, e.g., views, integrity constraints, assertions, data definition, aggregation, and coalescing. The language supports both point and interval-based semantics, with intervals being preserved by default.

We emphasize the fact that it is possible to control time-related processing via semantically defined statement modifiers. Not only is this possible but it is also preferable over syntactic extensions because statement modifiers ensure comprehensive availability of temporal functionality. We are aware of no other approaches that achieve comprehensive temporal support using language extensions that are essentially independent of the complexity of the underlying non-temporal language. The same (simple) statement modifier may be applied to an arbitrarily complex query to yield built-in temporal processing.

The paper illustrates how to define the semantics of a temporal statement modifier-extended SQL in terms of the semantics of SQL and a mapping from SQL to relational algebra. For this purpose, valid-time, transaction-time, and bitemporal counterparts of the standard relational algebra were defined.

The final step was to study properties of the temporal statement modifiers. Specifically, we verified that the extended SQL satisfies the requirements posed at the outset of the paper.

Several interesting directions for future research may be pointed out. First, the approach may be generalized to other "dimensions," such as those found in spatial databases, leading to spatio-temporal databases. It also appears promising to study how the proposed concepts generalize to databases annotated with other types of multiple orthogonal dimensions, e.g., those found in data warehousing. In doing so, an important challenge is to provide solutions that are general and yet succeed in supporting well the semantics associated with the specific dimensions. It may also prove interesting to generalize statement modifiers to multi-dimensional frameworks.

The notion of temporal upward compatibility makes the implicit assumption that the databases of existing DBMS's contain snapshot data and that a temporal dimension is added to data when the DBMS is replaced with a temporal DBMS. However, this scenario is not exhaustive. Rather, it may be observed that a wide variety of existing databases record time-varying data using regular attributes. The ability to use the novel features of the temporal DBMS depend on the time-varying data being recorded using the designated timestamp attributes. How to (semi-automatically) migrate application code when the transition is made from using regular attributes for capturing temporal aspects to using timestamp attributs with built-in semantics is an open problem.

Yet another future direction is the study of efficient implementation techniques. The current prototype illustrates the feasibility of the language using a layered architecture [40]. This architecture can be used to identify bottlenecks of current DB technology with respect to temporal database applications. The findings may then prompt the development of new DBMS algorithms. This approach has already been pursued for coalescing [9].

## 8 Acknowledgements

We greatly appreciate the contributions of Renato Busatto, Robert Marti, Rick Snodgrass, and Andreas Steiner. Renato contributed to the formalization of interval preservation, bitemporal negation, and the proof of Theorem 1. Robert took part in early work on statement modifiers. Rick contributed significantly to the design of the temporal extension when we jointly designed a different extension of SQL that is being proposed for inclusion into the SQL standard. Andreas implemented an initial, running prototype for the language proposed to the SQL standardization committee.

This research was supported in part by the Danish Technical Research Council through grant 9700780 and by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, as a Networks Activity of the Training and Mobility of Researchers Programme, contract no. FMRX-CT96-0056.

## A Predicates and Functions on Timestamps

For completeness reason we give a brief overview of the constructs used for timestamp manipulation. In Table 6, $tp$ and $iv$, possibly indexed, denote a time point of type `TIMESTAMP` and a time duration of type `INTERVAL`, respectively. Also, $per$ is a shorthand for `PERIOD` $'tp_1 - tp_2'$ and $granule \in \{$`YEAR`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, `SECOND`$\}$ denotes a granularity.

## B Auxiliary Algebraic Operators

Table 7 defines the auxiliary operators that timeslice relations and turn timestamps into regular, explicit attributes. Unlike the other algebraic operators defined in this paper, these operators are overloaded to apply to valid-time, transaction-time, and bitemporal relations, meaning that the type of the argument relation determines the

| *Syntax* | *Semantics* |
|---|---|
| $\llbracket \texttt{PERIOD '}\texttt{tp}_1 - \texttt{tp}_2\texttt{'} \rrbracket_{temp}$ | $\texttt{TIMESTAMP'tp}_1\texttt{'}, \texttt{TIMESTAMP'tp}_2\texttt{'}$ |
| $\llbracket \texttt{FIRST}(\texttt{TIMESTAMP'tp}_1\texttt{'}, \texttt{TIMESTAMP'tp}_2\texttt{'}) \rrbracket_{temp}$ | $\min(tp_1, tp_2)$ |
| $\llbracket \texttt{LAST}(\texttt{TIMESTAMP'tp}_1\texttt{'}, \texttt{TIMESTAMP'tp}_2\texttt{'}) \rrbracket_{temp}$ | $\max(tp_1, tp_2)$ |
| $\llbracket \texttt{VTIME}(\texttt{r}) \rrbracket_{temp}$ | $\texttt{TIMESTAMP'r.VT\$S'}, \texttt{TIMESTAMP'r.VT\$E'}$ |
| $\llbracket \texttt{TTIME}(\texttt{r}) \rrbracket_{temp}$ | $\texttt{TIMESTAMP'r.TT\$S'}, \texttt{TIMESTAMP'r.TT\$E'}$ |
| $\llbracket \texttt{BEGIN}(\texttt{per}) \rrbracket_{temp}$ | $\llbracket \texttt{FIRST}(\llbracket per \rrbracket_{temp}) \rrbracket_{temp}$ |
| $\llbracket \texttt{END}(\texttt{per}) \rrbracket_{temp}$ | $\llbracket \texttt{LAST}(\llbracket per \rrbracket_{temp}) \rrbracket_{temp}$ |
| $\llbracket per_1 \ \texttt{PRECEDES} \ per_2 \rrbracket_{temp}$ | $\llbracket \texttt{END}(\texttt{per}_1) \rrbracket_{temp} < \llbracket \texttt{BEGIN}(\texttt{per}_2) \rrbracket_{temp}$ |
| $\llbracket per_1 \ \texttt{MEETS} \ per_2 \rrbracket_{temp}$ | $\llbracket \texttt{END}(\texttt{per}_1) \rrbracket_{temp} = \llbracket \texttt{BEGIN}(\texttt{per}_2) -^{granule} 1 \rrbracket_{temp}$ |
| $\llbracket per_1 \ \texttt{OVERLAPS} \ per_2 \rrbracket_{temp}$ | $\llbracket \texttt{END}(\texttt{per}_1) \rrbracket_{temp} \geq \llbracket \texttt{BEGIN}(\texttt{per}_2) \rrbracket_{temp} \wedge$ $\llbracket \texttt{END}(\texttt{per}_2) \rrbracket_{temp} \geq \llbracket \texttt{BEGIN}(\texttt{per}_1) \rrbracket_{temp}$ |
| $\llbracket per_1 \ \texttt{CONTAINS} \ per_2 \rrbracket_{temp}$ | $\llbracket \texttt{BEGIN}(\texttt{per}_2) \rrbracket_{temp} \geq \llbracket \texttt{BEGIN}(\texttt{per}_1) \rrbracket_{temp} \wedge$ $\llbracket \texttt{END}(\texttt{per}_2) \rrbracket_{temp} \leq \llbracket \texttt{END}(\texttt{per}_1) \rrbracket_{temp}$ |
| $\llbracket per + \texttt{INTERVAL'}iv\texttt{'} \rrbracket_{temp}$ | $\llbracket \texttt{BEGIN}(\texttt{per}) \rrbracket_{temp} + iv,$ $\llbracket \texttt{END}(\texttt{per}) \rrbracket_{temp} + iv$ |
| $\llbracket \texttt{INTERSECT}(per_1, per_2) \rrbracket_{temp}$ | $\max(\llbracket \texttt{BEGIN}(\texttt{per}_1) \rrbracket_{temp}, \llbracket \texttt{BEGIN}(\texttt{per}_2) \rrbracket_{temp}),$ $\min(\llbracket \texttt{END}(\texttt{per}_1) \rrbracket_{temp}, \llbracket \texttt{END}(\texttt{per}_2) \rrbracket_{temp})$ |
| $\llbracket \texttt{DURATION}(per, granule) \rrbracket_{temp}$ | $\llbracket \texttt{END}(\texttt{per}) \rrbracket_{temp} -^{granule} \llbracket \texttt{BEGIN}(\texttt{per}) \rrbracket_{temp})$ |

Table 6: Definition of Constructs for Timestamp Manipulation

| Function | Semantics if $r$ is a snapshot relation |
|---|---|
| $\tau^{vt}_{tp}(r)$ | $r$ if $tp = now$; undefined, otherwise |
| $\tau^{tt}_{tp}(r)$ | $r$ if $tp = now$; undefined, otherwise |

| Function | Semantics if $r$ is a valid-time relation |
|---|---|
| $\tau^{vt}_{tp}(r)$ | $\{\langle t \rangle \mid \exists VT\ (\langle t \Vert VT \rangle \in r \wedge VT\ overlaps\ tp)\}$ |
| $\tau^{tt}_{tp}(r)$ | $\{\langle t \Vert VT \rangle \mid \langle t \Vert VT \rangle \in r\}$ |
| $\delta^{vt}_{per}(r)$ | $\{\langle t \Vert VT \rangle \mid \exists VT'\ (\langle t \Vert VT' \rangle \in r \wedge VT'\ overlaps\ per \wedge VT = intersect(VT', per))\}$ |
| $\delta^{tt}_{per}(r)$ | $\{\langle t \Vert VT \rangle \mid \langle t \Vert VT \rangle \in r\}$ |
| $SN^{vt}(r)$ | $\{\langle t, VT \rangle \mid \langle t \Vert VT \rangle \in r\}$ |
| $SN^{tt}(r)$ | $\{\langle t \Vert VT \rangle \mid \langle t \Vert VT \rangle \in r\}$ |

| Function | Semantics if $r$ is a transaction-time relation |
|---|---|
| $\tau^{vt}_{tp}(r)$ | $\{\langle t \Vert TT \rangle \mid \langle t \Vert TT \rangle \in r\}$ |
| $\tau^{tt}_{tp}(r)$ | $\{\langle t \rangle \mid \exists TT\ (\langle t \Vert TT \rangle \in r \wedge TT\ overlaps\ tp)\}$ |
| $\delta^{vt}_{per}(r)$ | $\{\langle t \Vert TT \rangle \mid \langle t \Vert TT \rangle \in r\}$ |
| $\delta^{tt}_{per}(r)$ | $\{\langle t \Vert TT \rangle \mid \exists TT'\ (\langle t \Vert TT' \rangle \in r \wedge TT'\ overlaps\ per \wedge TT = intersect(TT', per))\}$ |
| $SN^{vt}(r)$ | $\{\langle t \Vert TT \rangle \mid \langle t \Vert TT \rangle \in r\}$ |
| $SN^{tt}(r)$ | $\{\langle t, TT \rangle \mid \langle t \Vert TT \rangle \in r\}$ |

| Function | Semantics if $r$ is a bitemporal relation |
|---|---|
| $\tau^{vt}_{tp}(r)$ | $\{\langle t \Vert TT \rangle \mid \exists VT\ (\langle t \Vert VT, TT \rangle \in r \wedge VT\ overlaps\ tp)\}$ |
| $\tau^{tt}_{tp}(r)$ | $\{\langle t \Vert VT \rangle \mid \exists TT\ (\langle t \Vert VT, TT \rangle \in r \wedge TT\ overlaps\ tp)\}$ |
| $\delta^{vt}_{per}(r)$ | $\{\langle t \Vert VT, TT \rangle \mid \exists VT'\ (\langle t \Vert VT', TT \rangle \in r \wedge VT'\ overlaps\ per \wedge VT = intersect(VT', per))\}$ |
| $\delta^{tt}_{per}(r)$ | $\{\langle t \Vert VT, TT \rangle \mid \exists TT'\ (\langle t \Vert VT, TT' \rangle \in r \wedge TT\ overlaps\ per \wedge TT = intersect(TT', per))\}$ |
| $SN^{vt}(r)$ | $\{\langle t, VT \Vert TT \rangle \mid \langle t \Vert VT, TT \rangle \in r\}$ |
| $SN^{tt}(r)$ | $\{\langle t, TT \Vert VT \rangle \mid \langle t \Vert VT, TT \rangle \in r\}$ |

Table 7: Timeslice and Snapshot Operators

operation to be performed. This property was exploited to concisely define the semantics of core statements, in Table 2.

The functions have variants for both valid and transaction time. For example, the valid-time version of the first timeslice operation, $\tau^{vt}_{tp}$, selects all tuples in the argument relation with a timestamp that overlaps time point $tp$. The time dimension used in this selection is not present in the result relation. If valid time is not supported by the relation, the function degenerates to the identity function. The second timeslice operation, $\delta_{per}$, returns all argument tuples that overlap with period $per$. The timestamp of a result tuple is the intersection of $per$ with the tuple's original timestamp. The snapshot operation $SN$ turns a time dimension into an explicit attribute. This operation is not needed at the implementation level where all attributes are explicit.

## C The Bitemporal Relational Algebra

As the valid-time algebra was a natural generalization of the relational algebra, so is the bitemporal algebra a natural generalization of the valid-time algebra, and it satisfies the same snapshot reducibility properties as the valid-time algebra; it only differs from this algebra in that it deals with bitemporal rectangles rather than with periods. Bitemporal selection, projection, set union, and Cartesian product (see Figure 7) are straightforward extensions.

$$\begin{aligned}
\sigma^{bi}_{c}(r) &\triangleq \{\langle t \| VT, TT\rangle \mid \langle t \| VT, TT\rangle \in r \wedge c(\langle t \| VT, TT\rangle)\}\\
\pi^{bi}_{f}(r) &\triangleq \{\langle t_1 \| VT, TT\rangle \mid \exists t_2\ (\langle t_2 \| VT, TT\rangle \in r \wedge t_1 = f(\langle t_2 \| VT, TT\rangle))\}\\
r_1 \cup^{bi} r_2 &\triangleq \{\langle t \| VT, TT\rangle \mid \langle t \| VT, TT\rangle \in r_1 \vee \langle t \| VT, TT\rangle \in r_2\}\\
r_1 \times^{bi} r_2 &\triangleq \{\langle \langle t_1, VT_1, TT_1\rangle \circ \langle t_2, VT_2, TT_2\rangle \| VT, TT\rangle \mid\\
&\qquad \langle t_1 \| VT_1, TT_1\rangle \in r_1 \wedge \langle t_2 \| VT_2, TT_2\rangle \in r_2 \wedge\\
&\qquad VT = intersect(VT_1, VT_2)\ \wedge TT = intersect(TT_1, TT_2) \wedge\\
&\qquad VT_1\, overlaps\ VT_2 \wedge TT_1\, overlaps\ TT_2\}\\
r_1 \setminus^{bi} r_2 &\triangleq \{\langle t \| VT, TT\rangle \mid \exists VT_1, TT_1 (\langle t \| VT_1, TT_1\rangle \in r_1 \wedge\\
&\qquad candidate\_tuple(t, VT, TT, VT_1, TT_1, r_2) \wedge\\
&\qquad non\_overlapping(t, VT, TT, r_2) \wedge\\
&\qquad unsplittable(t, VT, TT, VT_1, TT_1, r_2))
\end{aligned}$$

Figure 7: The Bitemporal Algebra

Bitemporal difference is substantially more complex. It is defined in terms of three auxiliary predicates, to be defined below. The idea behind the operator's definition is illustrated in Figure 8, where the large rectangle with the solid frame represents the time region of an $r_1$-tuple, and the black ones are rectangles associated with value-equivalent $r_2$-tuples. The result of the difference $r_1 \setminus^{bi} r_2$ is a set of value-equivalent tuples, one for each of the eleven white rectangles identified by the dashed and solid lines in combination.

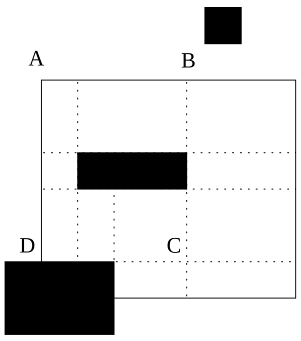


Figure 8: Bitemporal Difference

The so-called determining time lines associated with $r_2$-tuples play a crucial role in splitting $r_1$-tuples and thus in defining the result tuples. Determining time lines start at each vertex of an $r_2$-tuple, and they extend until they are blocked by a value-equivalent $r_2$-tuple or until they reach the border of the $r_1$-tuple. Before explaining the issues in more detail, it is convenient to first introduce some terminology. Each bitemporal tuple has associated a timestamp that encodes a rectangular region in the space spanned by transaction time and valid time. This region, we term the tuple's *time rectangle*, and the rectangle corners are termed *time vertices*. Their coordinates are the tuple's *time coordinates* which thus correspond to the tuple's transaction and valid time. The rectangle sides are *time edges* Finally, a *determining time line* is a vertical or horizontal line segment that originates from some time vertex. We omit the modifier "time" from these terms when no confusion results.

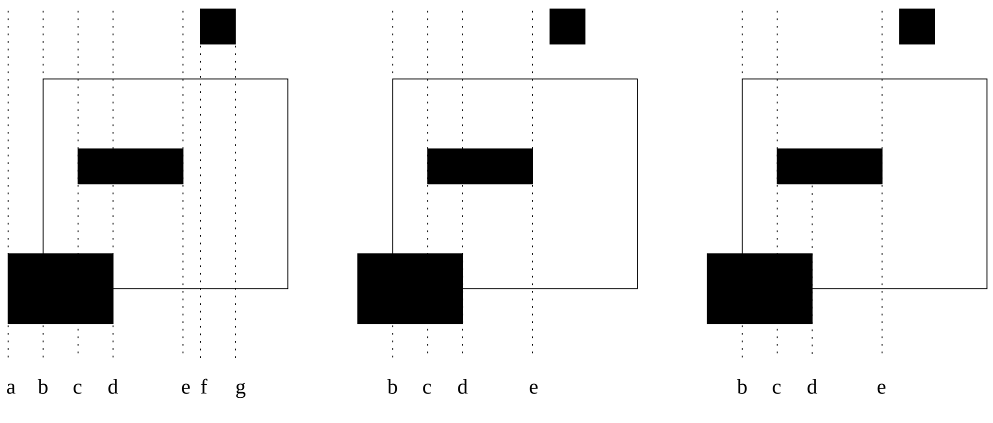
The definition in Figure 7 identifies three requirements to a result tuple $X$.

1. The time coordinates of $X$ are derived either from the time coordinates of a (value-equivalent) $r_1$-tuple, or from (value-equivalent) $r_2$-tuples that satisfy two restrictions:
   (a) They must temporally overlap with the $r_1$-tuple whose time rectangle contains the time rectangle of $X$, and
   (b) the time vertices of $X$ must have direct access to the originating $r_2$-tuple vertices, meaning that no value-equivalent $r_2$-tuple lies between originating and resulting vertices.
2. $X$ does not temporally overlap with any value-equivalent $r_2$-tuple.
3. No determining time lines defined by $r_2$-tuples that are value-equivalent to $X$ cross its time rectangle.

The first requirement, represented by the predicate $candidate\_tuple$, is defined as a conjunction of four subformulas, each of which constrains one of the time vertices of a tuple $\langle t \| VT, TT \rangle$ of $r_1 \setminus^{bi} r_2$.

$$
\begin{aligned}
&candidate\_tuple(t, VT, TT, VT_1, TT_1, r_2) \equiv \\
&\quad TT^- = TT_1^- \vee \\
&\quad \exists VT_2, TT_2(\langle t \| VT_2, TT_2 \rangle \in r_2 \wedge (TT^- = TT_2^- \vee TT^- = TT_2^+) \wedge \\
&\qquad TT_1^- \le TT^- < TT_1^+ \wedge VT_1^- < VT_2^+ \wedge VT_2^- < VT_1^+ \wedge \\
&\qquad \neg\exists VT_2, TT_2(\langle t \| VT_2, TT_2 \rangle \in r_2 \wedge TT_2^- < TT^- < TT_2^+ \wedge \\
&\qquad\quad (VT_2^+ < VT_2^+ \le VT^- \vee VT_2^- > VT_2^- \ge VT^+))) \wedge \\
&\quad TT^+ = TT_1^+ \vee \\
&\quad \exists VT_3, TT_3(\langle t \| VT_3, TT_3 \rangle \in r_2 \wedge (TT^+ = TT_3^- \vee TT^+ = TT_3^+) \wedge \\
&\qquad TT_1^- < TT^+ \le TT_1^+ \wedge VT_1^- < VT_3^+ \wedge VT_3^- < VT_1^+ \wedge \\
&\qquad \neg\exists VT_3, TT_3(\langle t \| VT_3, TT_3 \rangle \in r_2 \wedge TT_3^- < TT^+ < TT_3^+ \wedge \\
&\qquad\quad (VT_3^+ < VT_3^+ \le VT^- \vee VT_3^- > VT_3^- \ge VT^+))) \wedge \\
&\quad VT^- = VT_1^- \vee \\
&\quad \exists VT_4, TT_4(\langle t \| VT_4, TT_4 \rangle \in r_2 \wedge (VT^- = VT_4^- \vee VT^- = VT_4^+) \wedge \\
&\qquad VT_1^- \le VT^- < VT_1^+ \wedge TT_1^- < TT_4^+ \wedge TT_4^- < TT_1^+ \wedge \\
&\qquad \neg\exists VT_4, TT_4(\langle t \| VT_4, TT_4 \rangle \in r_2 \wedge VT_4^- < VT^- < VT_4^+ \wedge \\
&\qquad\quad (TT_4^+ < TT_4^+ \le TT^- \vee TT_4^- > TT_4^- \ge TT^+))) \wedge \\
&\quad VT^+ = VT_1^+ \vee \\
&\quad \exists VT_5, TT_5(\langle t \| VT_5, TT_5 \rangle \in r_2 \wedge (VT^+ = VT_5^- \vee VT^+ = VT_5^+) \wedge \\
&\qquad VT_1^- < VT^+ \le VT_1^+ \wedge TT_1^- < TT_5^+ \wedge TT_5^- < TT_1^+ \wedge \\
&\qquad \neg\exists VT_5, TT_5(\langle t \| VT_5, TT_5 \rangle \in r_2 \wedge VT_5^- < VT^+ < VT_5^+ \wedge \\
&\qquad\quad (TT_5^+ < TT_5^+ \le TT^- \vee TT_5^- > TT_5^- \ge TT^+)))
\end{aligned}
$$

The first two lines of the first conjunct have a *generative* purpose, since they identify a collection of candidate transaction-time start values for $\langle t \| VT, TT \rangle$. The left-most diagram below illustrates all such candidate values for the relation depicted in Figure 8:



Not all time lines originating from $r_2$-tuples provide suitable time coordinates, though. The subsequent three lines of the definition eliminate some of the undesirable ones. Specifically, the third line requires the time edge of an $r_2$-tuple that generates a candidate transaction time value to overlap the time rectangle of the relevant $r_1$-tuple. Time lines a, f and g above must then be dropped, as indicated in the middle diagram.

The fourth and fifth lines account for the blocking effect of $r_2$-tuples on candidate time lines. In the example, the upper part of line d is inadequate. The lines that meet all the restrictions, indicated in the right-most diagram, correspond to determining time lines for $r_1$ and $r_2$, provided they are confined to the time rectangle of the $r_1$-tuple.

The remaining conjuncts of $candidate\_tuple$ impose equivalent constraints on each of the other time coordinates of $r_1 \setminus^{bi} r_2$-tuples.

Next, the *non overlapping* of $r_1 \setminus^{bi} r_2$- and $r_2$-tuples is enforced by the following predicate.

$$\begin{aligned}
&non\_overlapping(t, VT, TT, r_2) \equiv \\
&\qquad \forall VT_2, TT_2(\langle t \| VT_2, TT_2\rangle \in r_2 \Rightarrow \\
&\qquad\qquad (VT^+ \leq VT_2^- \vee VT_2^+ \leq VT^- \vee TT^+ \leq TT_2^- \vee TT_2^+ \leq TT^-))
\end{aligned}$$

In the example, this predicate excludes the (aggregate) time rectangle ABCD in Figure 8, since it contains the rectangle of an $r_2$-tuple.

Finally, the partition of the time rectangle of an $r_1$-tuple must be *maximal* according to the previous restrictions, i.e., there should not be any additional determining time lines splitting a time rectangle of a tuple in $r_1 \setminus^{bi} r_2$. This can be ensured by requiring that, whenever the time edge of an $r_2$-tuple could originate an additional splitting line, there should exist another $r_2$-tuple blocking its effect:

$$\begin{aligned}
&unsplittable(\langle t \| VT, TT\rangle, \langle t \| VT_1, TT_1\rangle, r_2) \equiv \\
&\quad \forall VT_2, TT_2, tt(\langle t \| VT_2, TT_2\rangle \in r_2 \wedge \\
&\qquad (tt = TT_2^- \vee tt = TT_2^+) \wedge TT^- < tt < TT^+ \wedge \\
&\qquad (VT_1^- < VT_2^- < VT_1^+ \vee VT_1^- < VT_2^+ < VT_1^+) \Rightarrow \\
&\qquad\quad \exists VT_2, TT_2(\langle t \| VT_2, TT_2\rangle \in r_2 \wedge TT_2^- < tt < TT_2^+ \wedge \\
&\qquad\qquad (VT_2^+ < VT_2^+ \leq VT^- \vee VT_2^- > VT_2^- \geq VT^+))) \wedge \\
&\\
&\quad \forall VT_3, TT_3, vt(\langle t \| VT_3, TT_3\rangle \in r_2 \wedge \\
&\qquad (vt = VT_3^- \vee vt = VT_3^+) \wedge VT^- < vt < VT^+ \wedge \\
&\qquad (TT_1^- < TT_3^- < TT_1^+ \vee TT_1^- < TT_3^+ < TT_1^+) \Rightarrow \\
&\qquad\quad \exists VT_3, TT_3(\langle t \| VT_3, TT_3\rangle \in r_2 \wedge VT_3^- < vt < VT_3^+ \wedge \\
&\qquad\qquad (TT_3^+ < TT_3^+ \leq TT^- \vee TT_3^- > TT_3^- \geq TT^+)))
\end{aligned}$$

Hence, ABCD in the figure also does not qualify as the time rectangle of an $r_1 \backslash^{bi} r_2$-tuple because various (unblocked) determining time lines split it into seven rectangles.

Finally, we define coalescing of bitemporal relations. Transaction-time coalescing, $coal_{tt}^{bi}$ guarantees maximal transaction-time periods and is illustrated in Figure 9. The five white rectangles illustrate the times of five value-equivalent

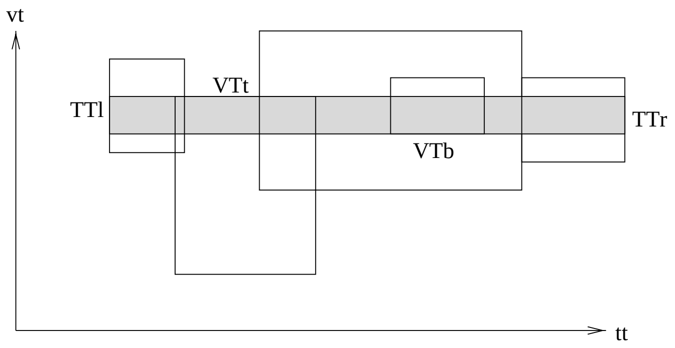


Figure 9: Transaction Time Coalescing of a Bitemporal Relation

tuples in the uncoalesced relation. The gray tuple is one of the tuples resulting from coalescing. (Transaction-time coalescing ensures maximal expansion in the transaction-time dimension and yields no coalescing in the valid-time dimension.)

Formally, coalescing is defined as follows.

$$
\begin{array}{l}
coal_{tt}^{bi}(r) \quad \stackrel{\triangle}{=} \quad \{\langle t \| VT, TT\rangle \mid \\
\quad \exists VT_1, TT_r(\langle t \| VT_1, TT_r\rangle \in r \wedge TT^+ = TT_r^+) \wedge \\
\quad \exists VT_2, TT_l(\langle t \| VT_2, TT_l\rangle \in r \wedge TT^- = TT_l^-) \wedge \\
\quad \exists VT_t, TT_3(\langle t \| VT_t, TT_3\rangle \in r \wedge \\
\qquad (VT^+ = VT_t^- \vee VT^+ = VT_t^+) \wedge TT_3^- < TT^+ \wedge TT_3^+ > TT^-) \wedge \\
\quad \exists VT_b, TT_4(\langle t \| VT_b, TT_4\rangle \in r \wedge \\
\qquad (VT^- = VT_b^+ \vee VT^- = VT_b^-) \wedge TT_4^- < TT^+ \wedge TT_4^+ > TT^-) \wedge \\
\quad \neg\exists VT_5, TT_5(\langle t \| VT_5, TT_5\rangle \in r \wedge \\
\qquad (VT^- < VT_5^+ < VT^+ \vee VT^- < VT_5^- < VT^+) \wedge \\
\qquad TT_5^- \leq TT^+ \wedge TT_5^+ \geq TT^-) \wedge \\
\quad \forall VT_6, TT_6(\langle t \| VT_6, TT_6\rangle \in r \wedge \\
\qquad TT^- \leq TT_6^- < TT^+ \wedge VT_6^- \leq VT^- \wedge VT_6^+ \geq VT^+ \Rightarrow \\
\qquad\qquad\qquad \exists VT_7, TT_7(\langle t \| VT_7, TT_7\rangle \in r \wedge TT_7^- < TT_6^- \leq TT_7^+ \wedge \\
\qquad\qquad\qquad\qquad VT_7^- \leq VT^- \wedge VT_7^+ \geq VT^+)) \wedge \\
\quad \neg\exists VT_8, TT_8(\langle t \| VT_8, TT_8\rangle \in r \wedge \\
\qquad (TT_8^- < TT^- \leq TT_8^+ \vee TT_8^- \leq TT^+ < TT_8^+) \wedge \\
\qquad VT_8^- < VT^+ \wedge VT_8^+ > VT^-)\}
\end{array}
$$

In the first two lines we search for two tuples defining the transaction-time start ($TT_l^-$) and the transaction-time end ($TT_r^+$) of a coalesced tuple. In lines 3 and 4, we do the same for valid-time start and end. Lines 5 and 6 ensure that no coalescing in the valid-time dimension is done, i.e., the extension in the valid-time dimension is as small as possible. Lines 7 to 9 ensure that there are no holes, i.e., *all* tuples with a transaction-time start contained in the final maximal transaction time must be covered by another tuple. The last two lines ensure that we get maximal extensions in transaction time, i.e., that no tuple exists that could possibly extend the tuple further.

Valid-time coalescing of a bitemporal relation $r$, $coal_{vt}^{bi}(r)$, follows the same principle, the only difference being that the roles of valid and transaction time are reversed. The definition is thus omitted.

## D Proof of Theorem 1

To prove Theorem 1, we consider each equivalence in turn. The two sides of the equivalence for selection are defined as follows.

$$\begin{aligned}
\tau_{tp}^{vt}(\sigma_c^{vt}(r)) &= \{t \mid \langle t \| VT\rangle \in r \wedge c(\langle t, VT\rangle) \wedge VT\ overlaps\ tp\}\\
\sigma_c(\tau_{tp}^{vt}(r)) &= \{t \mid \langle t \| VT\rangle \in r \wedge VT\ overlaps\ tp \wedge c(t)\}
\end{aligned}$$

To show that these definitions are equivalent, we first exploit the commutativity of conjunction to rewrite "$VT\ overlaps\ tp \wedge c(t)$" to "$c(t) \wedge VT\ overlaps\ tp$." What remains is to prove that $c(\langle t, VT\rangle)$ and $c(t)$ are equivalent. The same predicate $c$ occurs on both sides of the equality, and since the formulation of the theorem disallows the use of $VT$ in predicate $c$, the equality and thus the first equivalence follows.

The equivalence for projections follows similarly.

$$\begin{aligned}
\tau_{tp}^{vt}(\pi_f^{vt}(r)) &= \{t_1 \mid \exists t_2(\langle t_2 \| VT\rangle \in r \wedge t_1 = f(\langle t_2, VT\rangle)) \wedge VT\ overlaps\ tp\}\\
\pi_f(\tau_{tp}^{vt}(r)) &= \{t_1 \mid \exists t_2(\langle t_2 \| VT\rangle \in r \wedge VT\ overlaps\ tp \wedge t_1 = f(t_2))\}
\end{aligned}$$

The only difference with respect to selection is that we are dealing with a projection function, not a selection predicate. Similarly to before, we first commute two terms and then observe that $VT$ may be omitted as an argument of $f$ because the use of $f$ is disallowed in the theorem, meaning that $f(\langle t_2, VT\rangle)$ and $f(t_2)$ are equivalent.

Considering the union operators, we once again apply the definitions of the operators involved to the two sides.

$$\begin{aligned}
\tau_{tp}^{vt}(r_1 \cup^{vt} r_2) &= \{t \mid (\langle t \| VT\rangle \in r_1 \vee \langle t \| VT\rangle \in r_2) \wedge VT\ overlaps\ tp\}\\
\tau_{tp}^{vt}(r_1) \cup \tau_{tp}^{vt}(r_2) &= \{t \mid (\langle t \| VT\rangle \in r_1 \wedge VT\ overlaps\ tp) \vee\\
&\qquad (\langle t \| VT\rangle \in r_2 \wedge VT\ overlaps\ tp)\}
\end{aligned}$$

Transforming the first formula into disjunctive normal form proves the equivalence.

The equivalence involving the Cartesian products is somewhat more complicated to prove.

$$\pi^{-}_{r_1.VT,r_2.VT}(\tau^{vt}_{tp}(r_1 \times^{vt}_{c} r_2)) = \{t_1 \circ t_2 \mid \langle t_1 \| VT_1\rangle \in r_1 \wedge \langle t_2 \| VT_2\rangle \in r_2 \wedge$$
$$VT_1\, overlaps\, VT_2 \wedge VT = intersect(VT_1, VT_2) \wedge VT\, overlaps\, tp\}$$
$$\tau^{vt}_{tp}(r_1) \times_c \tau^{vt}_{tp}(r_2) =$$
$$\{t_1 \circ t_2 \mid \langle t_1 \| VT_1\rangle \in r_1 \wedge VT_1\, overlaps\, tp \wedge \langle t_2 \| VT_2\rangle \in r_2 \wedge VT_2\, overlaps\, tp\}$$

After the usual initial reordering of the terms of the formula, we are left with the proof of the equivalence between

"$VT_1\, overlaps\, VT_2 \wedge VT = intersect(VT_1, VT_2) \wedge VT\, overlaps\, tp$"

and

"$VT_1\, overlaps\, tp \wedge VT_2\, overlaps\, tp$."

We consider each formula in turn.

$$VT_1\, overlaps\, VT_2 \wedge VT = intersect(VT_1, VT_2) \wedge VT\, overlaps\, tp$$
$$\Downarrow$$
$$\text{(elimination of } VT)$$
$$\Downarrow$$
$$VT_1\, overlaps\, VT_2 \wedge intersect(VT_1, VT_2)\, overlaps\, tp$$
$$\Downarrow$$
$$\text{(replace periods with points, cf. Table 6)}$$
$$\Downarrow$$
$$VT_1^+ > VT_2^- \wedge VT_2^+ > VT_1^- \wedge$$
$$\max(VT_1^-, VT_2^-) < tp \wedge \min(VT_1^+, VT_2^+) > tp$$
$$\Downarrow$$
$$(\max(A, B) < C \equiv A < C \wedge B < C)$$
$$(\min(A, B) > C \equiv A > C \wedge B > C)$$
$$\Downarrow$$
$$VT_1^+ > VT_2^- \wedge VT_2^+ > VT_1^- \wedge$$
$$VT_1^- < tp \wedge VT_2^- < tp \wedge VT_1^+ > tp \wedge VT_2^+ > tp$$

Next we rewrite the second formula.

$$VT_1\, overlaps\, tp \wedge VT_2\, overlaps\, tp$$
$$\Downarrow$$
$$\text{(replace periods with points, cf. Table 6)}$$
$$\Downarrow$$
$$VT_1^- < tp \wedge VT_1^+ > tp \wedge VT_2^- < tp \wedge VT_2^+ > tp$$
$$\Downarrow$$
$$(A < C \wedge B > C \Rightarrow B > A)$$
$$\Downarrow$$
$$VT_1^- < tp \wedge VT_1^+ > tp \wedge VT_2^- < tp \wedge$$
$$VT_2^+ > tp \wedge VT_1^+ > VT_2^- \wedge VT_2^+ > VT_1^-$$

Apart from the order of the terms, the rewritten formulas are identical.

The final equivalence involves valid-time difference:

$$\begin{array}{rcl} \tau^{vt}_{tp}(r_1 \setminus^{vt} r_2) & = & \{t \mid \phi_1\} \\ \tau^{vt}_{tp}(r_1) \setminus \tau^{vt}_{tp}(r_2) & = & \{t \mid \phi_2\} \end{array}$$

where $\phi_1$ is defined as

$$\begin{array}{l} \exists VT, VT_1 \\ \quad \langle t \| VT_1 \rangle \in r_1 \wedge \\ \quad (\exists VT_2(\langle t \| VT_2 \rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\ \quad (\exists VT_3(\langle t \| VT_3 \rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\ \quad VT^- < VT^+ \wedge \\ \quad \neg\exists VT_4(\langle t \| VT_4 \rangle \in r_2 \ \wedge VT_4\, overlaps\ VT) \wedge \\ \quad VT\ overlaps\ tp\} \end{array}$$

and $\phi_2$ is defined as

$$\underbrace{\exists VT_1(\langle t \| VT_1 \rangle \in r_1 \wedge VT_1\, overlaps\ tp)}_{\psi_1} \wedge \underbrace{\neg\exists VT_2(\langle t \| VT_2 \rangle \in r_2 \wedge VT_2\, overlaps\ tp)}_{\psi_2} .$$

To prove the two sets equivalent, we have to show that the defining formulas are equivalent, i.e., $\phi_1 \equiv \phi_2$. We do so by proving two implications $\phi_1 \Rightarrow \phi_2$ and $\phi_1 \Leftarrow \phi_2$ in turn.

**A) ($\phi_1 \Rightarrow \phi_2$)** With $\phi_2 \equiv \psi_1 \wedge \psi_2$ we can rewrite $\phi_1 \Rightarrow \phi_2$ to $(\phi_1 \Rightarrow \psi_1) \wedge (\phi_1 \Rightarrow \psi_2)$ and prove each of the conjuncts in turn. The proof is based on the following theorems.

$$\begin{array}{llcl}
T_1 & \sigma_1 \Rightarrow \sigma_2 & \Rightarrow & (\sigma_1 \wedge \sigma_3) \Rightarrow \sigma_2 \\
T_2 & \sigma_1 \Rightarrow \sigma_2 & \Rightarrow & (\sigma_3 \wedge \sigma_1) \Rightarrow (\sigma_3 \wedge \sigma_2) \\
T_3 & \sigma_1 \Rightarrow (\sigma_2 \Rightarrow \sigma_3) & \equiv & (\sigma_1 \wedge \sigma_2) \Rightarrow \sigma_3 \\
T_4 & (\sigma_1 \Rightarrow \sigma_2) \wedge (\sigma_3 \Rightarrow \sigma_4) & \Rightarrow & (\sigma_1 \wedge \sigma_3) \Rightarrow (\sigma_2 \wedge \sigma_4) \\
T_5 & \sigma_1 \Rightarrow \forall v\ \sigma_2 & \equiv & \forall v(\sigma_1 \Rightarrow \sigma_2) \text{ if } v \text{ does not occur in } \sigma_1 \\
T_6 & X \subseteq Y & \equiv & \forall z(z \in X \Rightarrow z \in Y) \\
T_7 & VT_2 \subseteq VT_1 & \equiv & VT_1^- \leq VT_2^- \wedge VT_2^+ \leq VT_1^+ \\
T_8 & tp \in VT & \equiv & VT\ overlaps\ tp \\
T_9 & \multicolumn{3}{l}{\text{If } \Delta \vdash (\phi \Rightarrow \psi) \text{, then } \Delta \vdash ((\exists v \phi) \Rightarrow (\exists v \psi))} \\
T_{10} & \Delta \vdash \forall v \phi & \text{iff} & \Delta \vdash \phi
\end{array}$$

The first sub-proof starts with two formulas that are trivially true.

(1) $(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \Rightarrow$
$VT_1^- \leq VT^-$

(2) $(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \Rightarrow$
$VT^+ \leq VT_1^+$

We then apply the above theorems until $\phi_1 \Rightarrow \psi_1$ results. (With each intermediate formula, we indicate the formulas and theorems that were used deriving it.)

(3) $(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge$
$(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \Rightarrow$
$VT \subseteq VT_1$ (1), (2), $T_4$, $T_7$

(4) $(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge$
$(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge$
$VT^- < VT^+ \wedge$
$\neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \wedge VT_4\, overlaps\ VT) \Rightarrow$
$VT \subseteq VT_1$ (3), $T_1$

(5) $(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge$
$(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge$
$VT^- < VT^+ \wedge$
$\neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \wedge VT_4\, overlaps\ VT) \wedge$
$tp \in VT \Rightarrow$
$tp \in VT_1$ (4), $T_6$, $T_5$, $T_{10}$, $T_3$

(6) $\langle t \| VT_1\rangle \in r_1 \wedge$
$(\exists VT_2(\langle t \| VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge$
$(\exists VT_3(\langle t \| VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge$
$VT^- < VT^+ \wedge$
$\neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \wedge VT_4\, overlaps\ VT) \wedge$
$VT\, overlaps\ tp \Rightarrow$
$\langle t \| VT_1\rangle \in r_1 \wedge VT_1\, overlaps\ tp$ (5), $T_2$, $T_8$

The introduction of existential quantifiers ($T_9$) completes the proof.

To prove $\phi_1 \Rightarrow \psi_2$ we start out with the formula below:

$$\begin{aligned}&\neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \wedge VT_4\, overlaps\ VT) \wedge\\ &VT\, overlaps\ tp \Rightarrow\\ &\quad \neg\exists VT_4(\langle t \| VT_4\rangle \in r_2 \wedge VT_4\, overlaps\ tp)\end{aligned}$$

Again, it is easy to see that the formula is trivially true. Next, we apply $T_1$ followed by $T_9$ to get

$$\begin{array}{l}
\exists VT, VT_1 \\
\quad \langle t \| VT_1 \rangle \in r_1 \wedge \\
\quad (\exists VT_2(\langle t \| VT_2 \rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\
\quad (\exists VT_3(\langle t \| VT_3 \rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\
\quad VT^- < VT^+ \wedge \\
\quad \neg\exists VT_4(\langle t \| VT_4 \rangle \in r_2 \wedge VT_4\, overlaps\ VT) \wedge \\
\quad VT\ overlaps\ tp\ \Rightarrow \\
\qquad \neg\exists VT_4(\langle t \| VT_4 \rangle \in r_2 \wedge VT_4\, overlaps\ tp)
\end{array}$$

Renaming of a bound variable yields $\phi_1 \Rightarrow \psi_2$.

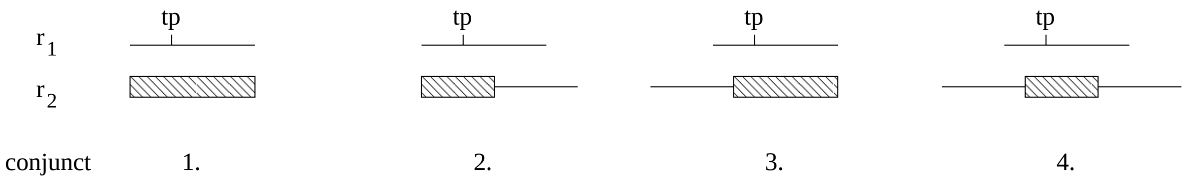
**B) ($\phi_1 \Leftarrow \phi_2$)** We prove $\phi_2 \Rightarrow \phi_1$ by reduction to absurdity, i.e., we show that $\neg\phi_1 \wedge \phi_2$ leads to a contradiction. We start with $\neg\phi_1$:

$$\begin{array}{l}
\forall VT, VT_1 \\
\quad (\langle t \| VT_1 \rangle \in r_1 \wedge \\
\quad (\exists VT_2(\langle t \| VT_2 \rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\
\quad (\exists VT_3(\langle t \| VT_3 \rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\
\quad VT^- < VT^+ \wedge \\
\quad VT\ overlaps\ tp \Rightarrow \\
\qquad \exists VT_4(\langle t \| VT_4 \rangle \in r_2 \wedge VT\ overlaps\ VT_4))
\end{array}$$

We first apply standard normalization rules [20, p.113] and quantifier elimination [13, p.49–58] to get $\phi_3$:

$$\begin{array}{l}
\forall VT_1(\langle t \| VT_1 \rangle \in r_1 \wedge VT_1^- \leq tp < VT_1^+ \Rightarrow \\
\qquad \exists VT_4(\langle t \| VT_4 \rangle \in r_2 \wedge VT_1^+ > VT_4^- \wedge VT_4^+ > VT_1^-)) \wedge \\
\forall VT_1, VT_2(\langle t \| VT_1 \rangle \in r_1 \wedge \langle t \| VT_2 \rangle \in r_2 \wedge VT_1^- \leq tp < VT_2^- \leq VT_1^+ \Rightarrow \\
\qquad \exists VT_4(\langle t \| VT_4 \rangle \in r_2 \wedge VT_2^- > VT_4^- \wedge VT_4^+ > VT_1^-)) \wedge \\
\forall VT_1, VT_2(\langle t \| VT_1 \rangle \in r_1 \wedge \langle t \| VT_2 \rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \leq tp < VT_1^+ \Rightarrow \\
\qquad \exists VT_4(\langle t \| VT_4 \rangle \in r_2 \wedge VT_1^+ > VT_4^- \wedge VT_4^+ > VT_2^+)) \wedge \\
\forall VT_1, VT_2, VT_3(\langle t \| VT_1 \rangle \in r_1 \wedge \\
\quad \langle t \| VT_2 \rangle \in r_2 \wedge \langle t \| VT_3 \rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \leq tp < VT_3^- \leq VT_1^+ \Rightarrow \\
\qquad \exists VT_4(\langle t \| VT_4 \rangle \in r_2 \wedge VT_3^- > VT_4^- \wedge VT_4^+ > VT_2^+))
\end{array}$$

Each conjunct of $\phi_3$ is represented in the diagram below. The solid lines represent the (times of the) first part of a conjunct, i.e., the part before the implication, whereas the grey rectangles indicate the time range that must be overlapped by yet another $r_2$-tuple (the second part of the conjunct, i.e., the part that follows the implication).



From $\psi_1$ and the first conjunct of $\phi_3$, it follows that a) there is an $r_1$-tuple $x_1 = \langle t \| VT_1 \rangle$ such that $tp \in VT_1$ and b) there is an $r_2$-tuple that temporally overlaps with $x_1$. Since, according to $\psi_2$ no $r_2$-tuple contains $tp$, the overlap must be of the form depicted in either the second or third diagram. Both cases imply that yet another $r_2$ tuple exists that overlaps the time period indicated by the gray rectangle. Depending on the time period of $r_2$, we end up with a situation represented by diagram two, three, or four. (Because of $\psi_2$, another overlapping is impossible.) Whatever situation it would imply yet another $r_2$ tuple which is (timely) closer to $tp$. No finite relation $r_2$ can fulfill this requirement.

## References

[1] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, December 1986.

[2] M. H. Böhlen, R. Busatto, and C. S. Jensen. Point-Versus Interval-Based Temporal Data Models. In *Proceedings of the 14th International Conference on Data Engineering*, Orlando, Florida, February, to appear 1998.

[3] J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Notions of Upward Compatibility of Temporal Query Languages. *Wirtschaftsinformatik*, 39(1):25–34, February 1997.

[4] G. Bhargava and S. K. Gadia. Relational Database Systems with Zero Information Loss. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):76–87, February 1993.

[5] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating the Completeness of TSQL2. In *Recent Advances in Temporal Databases, International Workshop on Temporal Databases*, pages 153–172, Zürich, Switzerland, September 1995. Springer, Berlin.

[6] M. H. Böhlen, C. S. Jensen, and B. Skjellaug. Spatio-Temporal Database Support for Legacy Applications. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, Marriott Marquis, Atlanta, Georgia, February, submitted 1998.

[7] M. Böhlen and R. Marti. On the Completeness of Temporal Database Query Languages. *Proceedings of the First International Conference on Temporal Logic*, pages 283–300, July 1994.

[8] M. Böhlen. *Managing Temporal Knowledge in Deductive Databases*. PhD thesis, Departement für Informatik, ETH Zürich, Switzerland, 1994.

[9] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In T. M. Vijayaraman, A. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the Twenty-second International Conference on Very Large Data Bases*, pages 180–191. Morgan Kaufmann Publishers, Inc., Mumbai (Bombay), India, September 1996.

[10] J. Clifford, A. Croker, and A. Tuzhilin. On the Completeness of Query Languages for Grouped and Ungrouped Historical Data Models. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 496–533. Benjamin/Cummings Publishing Company, 1993.

[11] J. Celko. *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, 1995.

[12] S. Ceri and G. Gottlob. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, April 1985.

[13] C. C. Chang and H. J. Keisler. *Model Theory*. North-Holland, Amsterdam, 3 edition, 1990.

[14] S. K. Gadia. Weak Temporal Relations. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1986.

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

[16] S. K. Gadia and G. Bhargava. SQL-like Seamless Query of Temporal Data. In R. T. Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, Texas, June 1993.

[17] 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.

[18] 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.

[19] C. S. Jensen, M. D. Soo, and R. T. Snodgrass. Unifying Temporal Models via a Conceptual Model. *Information Systems*, 19(7):513–547, 1994.

[20] J. W. Lloyd. *Logic Programming*. Symbolic Computation, Springer Verlag, Berlin, 2nd edition, 1987.

[21] N. A. Lorentzos and Y. G. Mitsopoulos. SQL Extension for Interval Data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), May/June 1997.

[22] L. E. McKenzie and R. T. Snodgrass. Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4):501–543, December 1991.

[23] J. Melton and A. R. Simon. *Understanding the new SQL: A Complete Guide*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[24] S. B. Navathe and R. Ahmed. TSQL - A Language Interface for History Databases. In *Proceedings of the Conference on Temporal Aspects in Information Systems*, pages 113–128. AFCET, May 1987.

[25] S. B. Navathe and R. Ahmed. A Temporal Relational Model and a Query Language. *Information Systems*, 49(2):147–175, 1989.

[26] 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.

[27] S. Nair and S. Gadia. Algebraic Optimization in a Relational Model for Temporal Databases. In R. T. Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, Texas, June 1993.

[28] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1985.

[29] N. Sarda. Algebra and Query Language for a Historical Data Model. *IEEE Computer Journal*, 33(1):11–18, February 1990.

[30] 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.

[31] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Transaction Time to SQL/Temporal. ANSI X3H2-96-152r, ISO–ANSI SQL/Temporal Change Proposal, ISO/IEC JTC1/SC21/WG3 DBL MCI-143, May 1996.

[32] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding Valid Time to SQL/Temporal. ANSI X3H2-96-151r1, ISO–ANSI SQL/Temporal Change Proposal, ISO/IEC JTC1/SC21/WG3 DBL MCI-142, May 1996.

[33] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. *Transitioning Temporal Support in TSQL2 to SQL3*. Proceedings of the Dagstuhl Seminar on Temporal Databases. to appear 1998.

[34] B. Schueler. Update reconsidered. In G. M. Nijssen, editor, *Architecture and Models in Data Base Management Systems*. North Holland Publishing Co., 1977.

[35] M. D. Soo, C. J. Jensen, and R. T. Snodgrass. An Algebra for TSQL2. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 27, pages 505–546. Kluwer Academic Publishers, 1995.

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

[37] R. T. Snodgrass. Temporal Databases: Status and Research Directions. *ACM SIGMOD Record*, 19(4):83–89, December 1990.

[38] R. T. Snodgrass. An Overview of TQuel. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 6, pages 141–182. Benjamin/Cummings Publishing Company, 1993.

[39] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, 1995.

[40] K. Torp, C. S. Jensen, and M. H. Böhlen. Layered Implementation of Temporal DBMSs-Concepts and Techniques. In *Proceedings of the Fifth International Conference On Database Systems For Advanced Applications, DASFAA '97*, Melbourne, Australia, to appear April 1997.

[41] D. C. Tsichritzis and F. H. Lochovsky. Data models. In *Software Series*. Prentice-Hall, 1982.

[42] D. Toman and D. Niwiński. First-Order Queries over Temporal Databases Inexpressible in Temporal Logic. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology — EDBT'96, 5th International Conference on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 307–324. Springer, March 1996.

[43] G. Wiederhold. How to Write a Schema for a Time Oriented Medical Record Data Bank. Technical report, Standford University, 1973.