Temporal Statement Modifiers

MICHAEL H. BÖHLEN and CHRISTIAN S. JENSEN
Aalborg University
and
RICHARD T. SNODGRASS
University of Arizona

A wide range of database applications manage time-varying data. Many temporal query languages have been proposed, each one the result of many carefully made yet subtly interacting design decisions. In this article we advocate a different approach to articulating a set of requirements, or desiderata, that directly imply the syntactic structure and core semantics of a temporal extension of an (arbitrary) nontemporal query language. These desiderata facilitate transitioning applications from a nontemporal query language and data model, which has received only scant attention thus far.

The paper then introduces the notion of *statement modifiers* that provide a means of systematically adding temporal support to an existing query language. Statement modifiers apply to all query language statements, for example, queries, cursor definitions, integrity constraints, assertions, views, and data manipulation statements. We also provide a way to systematically add temporal support to an existing implementation. The result is a temporal query language syntax, semantics, and implementation that derives from first principles.

We exemplify this approach by extending SQL-92 with statement modifiers. This extended language, termed ATSQL, is formally defined via a denotational-semantics-style mapping of temporal statements to expressions using a combination of temporal and conventional relational algebraic operators.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query languages*; *Data description languages* (DDL); *Data manipulation languages* (DML); H.2.4 [**Database Management**]: Systems—*Relational databases*; *Query processing*

General Terms: Languages, Theory

Additional Key Words and Phrases: ATSQL, statement modifiers, temporal databases

This research was supported in part by the Swiss National Science Foundation, the Danish Technical Research Council grant 9700780 and by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development contract FMRX-CT96-0056, by a grant from the Nykredit Corporation, and by National Science Foundation grants IRI-9632569 and IRI-9811406.
Authors' addresses: M. H. Böhlen and C. S. Jensen, Department of Computer Science, Aalborg University, Aalborg Øst, DK-9220, Denmark; email: boehlen@cs.auc.dk; csj@cs.auc.dk; R. T. Snodgrass, Department of Computer Science, University of Arizona, Tucson, AZ 85721-0077; email: rts@cs.arizona.edu.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 0362-5915/00/1200–0407 $5.00

1. INTRODUCTION

A wide variety of applications manage substantial amounts of time-varying data, including financial applications (portfolio management, budgeting, billing, accounting, and banking); record-keeping applications (personnel, medical-records, insurance policies, and inventory); travel applications (airline, train, and hotel reservations); project management applications (scheduling); and scientific applications (trend analysis). These numerous database applications manage substantial quantities of time-varying data. This has held true for as long as databases have been maintained [Wiederhold 1973; Snodgrass 1990]. Along with the continued improvement in storage technologies and new data-intensive applications such as decision support and data warehousing, old versions of data are retained longer in databases. This results in very large databases utilizing a prominent temporal dimension [Jensen and Snodgrass 1999].

When managing time-varying data, two temporal aspects attract special attention. The *valid time* of a database fact (e.g., a tuple) is the time that the fact was or will be true in the modeled reality. The *transaction time* of a database fact is the time when the fact is stored as current in the database. All database facts have a valid time and a transaction time, and we consider both of these; but a database relation is not required to capture either of these aspects. The terms *temporal* or *time-varying* are used when one or both valid time and transaction time are captured. For more specific situations, we use the terms *valid-time relation* for a relation that records valid time only; *transaction-time relation* for a relation that records transaction time only; and *bitemporal relation* for a relation that records both valid and transaction time. The term *nontemporal* (or *snapshot*) relation indicates a conventional relation that does not capture any temporal aspect [Jensen and Dyreson 1998].

Relational database technology provides little support for temporal data management, and is incapable of exploiting time dimension. In response to this unfulfilled potential, much work on temporal database management has been conducted over the past two decades, leading to a wide variety of data models and query languages and to numerous performance-enhancing implementation techniques [Zaniolo 1997]. Recent query languages (e.g., IXSQL [Lorentzos and Mitsopoulos 1997]; TempSQL [Gadia 1988]; and TSQL2 [Snodgrass 1995]) demonstrate that temporal application development may benefit substantially from built-in temporal support in the query language.

Providing temporal support directly in the query language and underlying DBMS is only half the answer, transitioning easily from existing technology to the new support is also important. A query language that facilitates this transition will be more successful than one that requires, say, a complete rewriting of existing applications. Fortunately, as we will show, it is possible to have the best of both worlds: powerful temporal support and a well-defined, straightforward migration path.

Existing temporal query languages, including those in whose design we have participated, have an unfortunate ad hoc feel to them. They contain numerous novel constructs, each with more or less justification. Each language is a confluence of many design decisions, which interact in subtle ways.

In this article we advocate a different approach to temporal query language design. We articulate desiderata that constrain and guide the design, so that, starting from an initial nontemporal query language (e.g., SQL-92 [Melton and Simon 1993]; SQL:1999 [Melton 1999]; ODMG [Cattell et al. 2000]), a syntactic structure and core semantics for a temporal query language are directly implied. The result is a language design that derives from first principles.

After presenting the desiderata, the article shows how to employ *statement modifiers* in the design of a temporal extension. For concreteness, we apply statement modifiers to SQL-92. In addition to queries (select statements), we also address data definition and modification statements, as well as integrity constraints. The language is described in two steps. First, we show how the desiderata shape the skeleton of a language. Second, we present a formal definition of the query language semantics 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 temporal and conventional 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 time periods associated with argument data and not on intermediate points, making evaluation granularity independent. We again emphasize that this general approach is amenable to any initial query language; we focus on the SQL-92 language to be specific and rigorous. Having defined the temporal extension based on statement modifiers, we then show that indeed its definition satisfies the desiderata.

The presentation is structured as follows. The next section formulates requirements for a temporal data model and query language. 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, semantics for the language, and shows how an implementation in terms of relational algebra can be extended to a temporal relational algebra. This provides a solid footing for the exploration of language properties—the topic of Section 5. Related research is explored in Section 6, and Section 7 summarizes and points to opportunities for future research. Appendices with detailed technical material complete the paper. TIGER, a prototype system that implements selected aspects of temporal SQL, is accessible via URL <www.cs.auc.dk/tiger>. Throughout this article we use snippets from the following application.

Table I. Desiderata Overview

Desiderata facilitating the transition to a temporal DBMS

upward compatibility Aim: to facilitate the migration of legacy code from an existing DBMS to a new temporal DBMS (Section 2.1).

temporal upward compatibility Aim: to facilitate the coexistence of legacy code and new temporal code, following migration, by allowing existing data to be rendered temporal without this affecting the operation of existing applications (Section 2.2).

S-reducibility Aim: to facilitate the reuse of the expertise of application developers, by offering temporal support that generalizes the non-temporal query language using a point-based view of a temporal database (Section 2.4).

Desiderata concerning systematic temporal support

S-reducibility described above; also yields a systematic approach to offering point-based temporal support.

extended S-reducibility Aim: to offer systematic temporal support, by ensuring reducibility even in statements that explicitly reference time (Section 2.5).

integrated support for point and interval-based views of data

interval preservation Aim: to provide maximal support for an interval-based view of data, by maximally respecting the intervals of argument tuples in point-based query language statements (Section 2.6).

nonrestrictiveness Aim: to provide maximal support for an interval-based view of data, by making it possible to treat interval-valued timestamps as normal values, and vice-versa (Section 2.7).

Example 1.1 Consider a human resources application that maintains a database of several SQL-92 relations recording who works in the company, their current salary and job assignment, as well as other current data. The application is inherently nontemporal: when someone receives a raise or is assigned to a different job, the modification is made in place, without retaining past data. We wish to migrate the application to the next release of the DBMS, which supports a temporal extension of SQL-92.

2. DESIDERATA FOR A TEMPORAL EXTENSION

The most prominent prospective users of temporal database technology are enterprises that manage large amounts of time-varying data. The challenge before us is two-fold: how to ensure that an enterprise can smoothly migrate from its current DBMS to a temporal DBMS, and how can that temporal DBMS actually deliver the promised systematic support for time-referenced data.

Table I summarizes desiderata that aim to ensure that these goals are achieved; the remainder of this section briefly shows how they, in concert, cover key language design issues.

Upward compatibility is well known throughout the software industry and is taken very seriously, for example, in SQL standardization. Specific to temporal databases, *temporal upward compatibility* requires that it be possible to render database relations temporal—to serve new temporal application code without impacting existing legacy code. The objective is to make it possible to gradually develop temporally-enabled applications without invalidating existing applications.

Having adopted a temporal DBMS, it is important that application developers be able to continue their work without an initial decrease in productivity. *S-reducibility* allows all existing SQL statements to be rendered temporal using a simple principle: a relation with a temporal dimension is perceived as a sequence of nontemporal relations to which regular SQL statements apply. A regular SQL statement can be rendered temporal by applying it individually to each snapshot state of a temporal relation and then combining the results into a single temporal relation.

The remaining desiderata are independent of legacy issues, and instead guide the design of the temporal extension itself. *Extended S-reducibility* ensures that all aspects of the extended query language where temporal support is meaningful should be given such support. *Interval preservation* and *nonrestrictiveness* aim to preserve intervals as much as possible, and permit the manipulation of interval-valued timestamps using built-in predicates and functions.

It should be noted that these desiderata are orthogonal to the performance properties of the migrated applications. We now formally define each in turn.

2.1 Upward Compatibility

With the new temporal DBMS, it is fundamentally important that *all* code work must *without modification*, and with exactly the same functionality as existing DBMS. Upward compatibility captures the essence of what is needed to make this possible. We assume that the interface of a DBMS is captured in a data model, and hence discuss the migration of application code using an existing data model rather than a new data model. We adopt the convention that a data model consists of two components, namely a set of data structures and a language for querying data structures [Tsichritzis and Lochovsky 1982]. For example, the relational model's central data structure is the relation, and the central user-level query language is SQL. Notationally, $M = (DS, QL)$ denotes a data model M, consisting of data structures DS and query language 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, that is, `INSERT`, `DELETE`, or `UPDATE`).

A (new) data model is *syntactically upward compatible* with another (legacy) data model if all the data structures and legal query expressions of the latter model are contained in the former model. If that is the case, all existing application code will remain syntactically correct.

Definition 2.1 (*Syntactic upward compatibility*). Let $M^x = (DS^x, QL^x)$ and $M^y = (DS^y, QL^y)$ be two data models. Model M^y is *syntactically upward compatible* with model M^x iff

$$\forall db^x \in DS^x(db^x \in DS^y) \text{ and } \forall s^x \in QL^x(s^x \in QL^y).$$

In addition, all queries expressible in the existing model must evaluate to the same results in the new model. For a query language expression s and an associated database db, both legal elements QL and DS of data model $M = (DS, QL)$, we let $\langle\langle s(db)M\rangle\rangle$ be the result of applying s to db in data model M. (This result captures all aspects of the interaction between an application and the DBMS, and thus includes result codes, changes to the data structures, cursor management, etc.)

Definition 2.2 (*Upward compatibility*). Let $M^x = (DS^x, QL^x)$ and $M^y = (DS^y, QL^y)$ be two data models. Model M^y is *upward compatible* with model M^x iff M^y is syntactically upward, compatible with M^x, and

$$\forall db^x \in DS^x(\forall s^x \in QL^x(\langle\langle s^x(db^x)\rangle\rangle_{M^x} = \langle\langle s^x(db^x)\rangle\rangle_{M^y})).$$

This guarantees that all existing queries compute the same results in the new system. (As we see in Section 2.3, this has strong implications for the implementation, down to the details of which error codes are to be returned.) Thus, the bulk of legacy application code is not affected by the transition to the new system.

Example 2.1 To illustrate upward compatibility (UC) in the context of SQL-92, consider the following statements.

```
CREATE TABLE Employee (ID INTEGER CONSTRAINT Employee_pk PRIMARY KEY, Name CHAR(30));
CREATE TABLE Salary (ID INTEGER, Amt INTEGER, FOREIGN KEY (ID) REFERENCES Employee);

SELECT E.Name, S.Amt FROM Employee AS E, Salary AS S WHERE E.ID = S.ID;

CREATE VIEW v AS SELECT * FROM Employee WHERE ID NOT IN (SELECT ID FROM Salary);
```

These are simple legacy SQL-92 statements that must be supported by any reasonable (temporal) extension of SQL-92; the semantics is dictated by SQL-92 [Melton and Simon 1993]. The first data definition statement defines a table with two columns. A column constraint states that `ID` is a primary key. The second statement defines another table with two columns. A table constraint makes `Salary.ID` reference `Employee.ID`. The select statement joins `Employee` and `Salary`. The fourth statement defines a view that returns all tuples in `Employee` that are not referenced by `Salary`.

By requiring that a temporal extension be a strict superset (i.e., only *adding nonmandatory* constructs and semantics), it is relatively easy to ensure that the temporal extension is upward compatible with SQL-92. Still, 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 extension because this string has already been used for the duration data type.[1]

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

Upward compatibility has one unintended ramification. Any extension that includes new reserved keywords violates upward compatibility, because legacy query language statements may have employed such keywords as identifiers. Under the semantics of the new model, such statements are illegal, but it is impractical to exclude new reserved keywords from temporal as well as nontemporal extensions. For example, SQL-92 added some 112 reserved keywords to the 115 reserved keywords of its predecessor, SQL-89.[2] To follow current practice and avoid being overly restrictive, we consider upward compatibility as satisfied even when new keywords are added.

2.2 Temporal Upward Compatibility

Upon adopting a temporal data model, the benefits of built-in temporal support are only realized incrementally by modifying existing application code or developing new application code that exploits temporal capabilities. Thus, a next step is to formulate desiderata that aim to ensure the 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 nontemporal. Now an application needs support for the temporal dimension of the data in one of the existing relations, and the existing nontemporal relation is then changed to a temporal one. It is undesirable to have to change the (legacy) application code that accesses the previously nontemporal relation. Therefore, we require that the existing applications on nontemporal relations must continue to work unmodified when nontemporal relations are rendered temporal. Intuitively, a query q must return the same result on an associated nontemporal database db as on the temporal counterpart of the database, $\mathcal{T}(db)$. Further, modification statements must ensure that subsequent queries return the expected results.

Definition 2.3 (*Temporal upward compatibility*). Let a temporal and a nontemporal 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 relation from nontemporal to temporal. 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

[2] Melton and Simon [1993] provide a list of ten items with incompatibilities in SQL-89 and SQL-92, with the keyword being one item.

constructs, the means of creating temporal tables, and the ability to apply SQL-92 queries and updates to temporal tables.

Example 2.2 To illustrate, we assume a temporal DBMS satisfying upward compatibility and TUC. Just to make things more interesting, assume for the moment that we have only the executable version of the human resources application from Example 2.1: we do not have access to the source code with embedded SQL.

Upward compatibility ensures that the application runs fine. Everything works as it did before.

Assume that the database state is as shown below.

Employee			Salary	
ID	Name		ID	Amt
1	Bob		1	20
3	Pam		3	40
4	Sarah			

We now render the database temporal by adding valid-time support to some of the underlying relations. The sole new construct, which is underlined, enables us to render a table temporal. (Alternatively, valid time could be captured by adding a special valid-time column, thus not altering the table type [Böhlen et al. 1998b].)

```
ALTER TABLE Employee ADD VT;
```

Let the current time be 7. Altering the `Employee` table associates with each row the timestamp 7 − NOW. (NOW is a special value that tracks the current time [Clifford et al. 1997].) The database is consistent since, at time 7, both integrity constraints are satisfied, that is, `Employee.ID` is a primary key and `Salary.ID` references `Employee.ID`.

Temporal upward compatibility ensures that the application still executes as it did before. The semantics must take into account that `Employee` is temporal, whereas `Salary` is still nontemporal. Specifically, queries that retrieved data still retrieve the same data, cursors work exactly as before, referential integrity-checking still applies as before, and so on. However, past states are retained automatically. We could implement new applications that used enhanced statements to query time-varying data; but the existing application would remain as is, without a single line of code needing to be changed.

Now let us assume that we do have the code for the application, which includes the following statements, executed at time 9.

```
INSERT INTO Employee VALUES (6, 'Tom');
DELETE FROM Employee WHERE ID = 4;

SELECT * FROM Employee
WHERE NOT EXISTS (SELECT * FROM Salary WHERE Salary.ID = Employee.ID);

SELECT * FROM v;
```

These statements are all legacy SQL-92 statements, but the semantics has changed because the underlying data structure, that is, table `Employee`, has changed. For example, the modification statements must adequately maintain the valid time of `Employee` [Bair et al. 1997]. The insert statement adds $\langle 6, \text{'Tom'} \| 9-\text{NOW}\rangle$ to `Employee`, and the delete statement changes $\langle 4, \text{'Sarah'} \| 1-\text{NOW}\rangle$ to $\langle 4, \text{'Sarah'} \| 1-8\rangle$. (The vertical double-bar "‖" separates the valid-time attribute *VT* from the nontemporal explicit attributes. We assume closed intervals.)

For the select and view statements, `Employee` must be restricted to the current state (which can be achieved in practice by recompiling the view definition). Both queries return ⟨6, 'Tom'⟩ because, at time 9, the only value of `Employee.ID` that does not occur in `Salary.ID` as well is 6, which we just inserted.

2.3 Implications of UC and TUC

Upward compatibility and temporal upward compatibility are concerned with statements in the temporal query language that are also in the initial nontemporal query language (i.e., *legacy statements*). Here we summarize the problems in implementing an upward compatible (in both senses) temporal extension of SQL-92.

Every statement *S* in (nontemporal) SQL-92, when evaluated on the temporal DBMS (TDBMS) being implemented, and referencing only nontemporal SQL-92 relations should have *exactly* the same semantics as a nontemporal DBMS. The semantics is well defined for the SQL-92 standard [Melton 1992], and includes the resulting tuples, all changes to the underlying relation (if the statement is a modification statement), cascading changes, checking of integrity constraints and assertions, invocation of defined triggers, impact on the transaction in which the statement executes, error conditions and codes returned to the application, and impact on cursors, etc.

Temporal upward compatibility provides a well-defined semantics for every statement *S* in (nontemporal) SQL-92, when evaluated on the TDBMS and referencing *temporal* relations. In interaction with the application, that semantics is defined in the SQL-92 standard, and includes, as before, the resulting tuples, impact on the transaction within which the statement executes, error conditions and codes, impact on the state of cursors, etc. For the application in which *S* occurs, there is *no* way in which the application can employ SQL-92 statements to detect whether an underlying relation is temporal because such statements have exactly the same effect on nontemporal and temporal relations.

The only difference in evaluating *S* in the TDBMS on a nontemporal versus a temporal relation is in the stored state of that relation, that is, a valid-time history is maintained. Such history can be seen in statements in the temporal extension of SQL-92, but, again, cannot be observed via legacy statements.

It is the responsibility of the TDBMS implementor to ensure that this correspondence is preserved at all times. Upward compatibility is easily ensured by retaining the syntax and implementation of the underlying nontemporal query language. As a first cut in supporting temporal upward compatibility, all operations that retrieve data from temporal relations should include an additional predicate that valid and transaction timestamps must now overlap, all operations that logically delete tuples should set the valid and transaction end times to now, and all operations that insert tuples should set the valid and transaction start times to now. The details depend on the underlying query language and DBMS implementation being extended.

There is another class of applications that by using date or time attributes we would like to migrate, those that store time-varying information in conventional relations. There are as many ways of doing this as there are programmers. (We have examined a diverse spectrum of approaches elsewhere [Snodgrass 2000].) Upward compatibility ensures that such applications still run on the temporal DBMS. However, temporal upward compatibility is less relevant in this situation because the relations are already storing time-varying information, though this is not apparent to the underlying DBMS (which has no notion of time-varying information). What is needed is a way to convert a relation with datetime attributes into a temporal relation. Such a conversion by necessity involves extended statements, and is discussed in that context in Section 2.7.

Now that we have dealt with the semantics of statements from the underlying nontemporal query language, we turn to desiderata for enhanced statements, present only in the temporal query language.

2.4 S-Reducibility

Most temporal extensions introduce new language constructs that mirror existing constructs, while emphasizing the time component. For example, TOSQL [Ariav 1986] adds a WHILE clause, and TSQL [Navathe and Ahmed 1989] and TQuel [Snodgrass 1987] each add a WHEN clause, paralleling the WHERE clause. Instead, we advocate utilizing the existing syntax and semantics and generalizing them to apply even when time is present. In this way, syntactically similar snapshot reducibility—called *S-reducibility*, for short—aims to protect the investment in programmer training and to ensure 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 nontemporal query language, for example, SQL-92.

S-reducibility states that the temporally extended data model's query language must offer, for each query in the nontemporal query language, a syntactically similar temporal query that is its "natural" generalization, in a precise technical sense. The goal is to make the semantics of temporal queries easily understandable in terms of the semantics of the corresponding SQL-92 queries on nontemporal relations. 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.

S-reducibility is based on the notion of snapshot reducibility. We use r and r^{bi} for denoting, respectively, a nontemporal and bitemporal relation instance. Similarly, db and db^{bi} are sets of nontemporal and bitemporal relation instances, respectively. The bitemporal timeslice operator $\tau^{M^{bi}, M}_{(c^{tt}, c^{vt})}$ (e.g., Schueler [1977] and Böhlen and Marti [1994]) 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 nontemporal 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.

Snapshot reducibility utilizes the so-called *point-based view*, which interprets a temporal relation as a collection of snapshots, each associated with a point in time. This view is illustrated in the following example.

Example 2.3 Assume schemas $P(A \| VT)$ and $Q(A \| VT)$ and instances $p(P) = \{\langle a \| 21-30 \rangle\}$ and $q(Q) = \{\langle a \| 23-27 \rangle\}$. Under the point-based view, p and q denote sequences of relations: $p_{21} = \ldots = p_{30} = \{\langle a \rangle\}$ and $q_{23} = \ldots = q_{27} = \{\langle a \rangle\}$. The index indicates the time of a state, that is, p_{21} is the state of relation p at time 21. States not shown are empty. The point-based view can be used to define a wide range of operations. For example, the point-based valid-time difference $r = p \setminus^{vt} q$ is defined as a sequence of regular differences on database states: $r_1 = p_1 \setminus q_1$, $r_2 = p_2 \setminus q_2$, etc. With the given instances, we get $r_{21} = r_{22} = r_{28} = r_{29} = r_{30} = \{\langle a \rangle\}$, which we write as $r = \{\langle a, 21-22 \rangle, \langle a, 28-30 \rangle\}$.

While this example examined a particular algebraic operator, relational difference, this notion can be expanded to statements in a query language, and indeed, to the query language as a whole.

Definition 2.4 (Snapshot reducibility) [Snodgrass 1987]. Let $M = (DS, QL)$ be a nontemporal 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

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

In other words, snapshot reducibility implies that for all query expressions q in the nontemporal 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}, making it possible for q^{bi} to be quite

different from q; so q^{bi} might be very involved even if q is not. This is undesirable when we would like to obtain a straightforward extension. Consequently, we require that q^{bi} and q be syntactically similar.

Definition 2.5 (*S-reducibility*). [Böhlen et al. 1995] Given nontemporal and bitemporal data models $M = (DS, QL)$ and $M^{bi} = (DS^{bi}, QL^{bi})$, data model M^{bi} is a *syntactically similar snapshot-reducible (S-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 nontemporal and temporal relations. In this case, the types of the argument relations determine the meaning of the construct. However, if temporal upward compatibility 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 S-reducibility satisfied, a reducible, temporal query evaluates to a result consistent with evaluating its 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, for example, 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 collection of snapshots. Explanations follow the example statements.

Example 2.4 We illustrate with a variety of sequenced statements from our human resources application.

```
SEQ VT SELECT * FROM Employee;

SEQ VT SELECT E.Name, S.Amt FROM Employee AS E, Salary AS S WHERE E.ID = S.ID;

SEQ VT
   SELECT ID FROM Employee AS E
   WHERE NOT EXISTS (SELECT * FROM Salary AS S WHERE E.ID = S.ID);

   CREATE TABLE Positions (ID INTEGER SEQ VT PRIMARY KEY, Position INTEGER) AS VT;
```

The first query simply returns all tuples together with their valid times—this corresponds to returning the content of `Employee` at each state (we requested the timestamp via `SEQ VT`). The remaining queries assume that table `Salary` has also been altered to become a valid-time table and that the database has the following contents.

Employee				Salary		
ID	Name	VTIME		ID	Amt	VTIME
1	Bob	5–8		1	20	4–10
3	Pam	1–3		3	20	6–9
3	Pam	4–12		4	20	6–9
4	Sarah	1–5				

The second query joins `Employee` and `Salary` at each state of the database. This amounts to the well-known temporal natural join [Jensen et al. 1994] and returns $\{\langle\text{'Bob'}, 20\|5-8\rangle\}$, $\langle\text{'Pam'}, 20\|6-9\rangle$. Conceptually, we get the result by evaluating 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). Similarly, the third query evaluates the query and the subquery, on each state of the database, thereby performing a variant of temporal difference. Again, the modifier `SEQ VT` tells the DBMS to compute the difference at each snapshot, and returns $\{\langle 3\|1-3\rangle, \langle 3\|4-5\rangle, \langle 3\|10-12\rangle, \langle 4\|1-5\rangle\}$. Again, we conceptually evaluate the enclosed statement on each state of the database. Computationally, we, for example, subtract the interval $6-9$ from the interval $4-12$ to get the intervals $4-5$ and $10-12$.

The last statement defines a table `Positions` and requires column `ID` to be a "temporal" primary key, that is, `ID` must be a primary key at each state (but not necessarily across states). Let `Positions` have the following contents.

Positions		
ID	Position	VTIME
1	101	3–6
1	204	10–17
2	306	4–8

The database is consistent because at each state `Positions.ID` is a primary key. Adding $\langle 1, 106\|7-9\rangle$ to `Positions` leaves the database in a consistent state, but adding $\langle 2, 106\|7-9\rangle$ violates the constraint, since `Positions.ID` would then not be a primary key at times 7 and 8.

These examples illustrate that S-reducible statements are easy to write and understand because they are simply conventional SQL statements with

the additional prefix SQL VT. Despite their natural semantics, these statements are very difficult to write without statement modifiers. A skilled SQL programmer would find it quite difficult to formulate even these simple examples in pure SQL [Snodgrass 2000].

2.5 Extended S-Reducibility

S-reducibility (Definition 2.5) is applicable only to queries of the underlying nontemporal query language, and does not extend to queries that explicitly reference the time.

Example 2.5 Consider these two similar queries.

```
SEQ VT
   SELECT E.ID
   FROM Employee AS E, Salary AS S
   WHERE E.ID = S.ID
   AND VTIME(E) OVERLAPS PERIOD '1990-2000'
```

```
SEQ VT
   SELECT E.ID, VTIME(S), VTIME(E)
   FROM Employee AS E, Salary AS S
   WHERE E.ID = S.ID
```

The query to the left constrains the temporal join to tuples in Employee with a valid time that overlaps the period 1990–2000. This condition cannot be evaluated on individual nontemporal relation states because the timestamp is not present in these states. Nevertheless, the temporal join itself can still be conceptualized as a nontemporal join evaluated on each snapshot, with an additional predicate. The query to the right computes a temporal join as well, but also returns the original valid times. Again, the semantics of this query falls outside of snapshot reducibility because the original valid times are not present in the nontemporal relation states.

DBMSs generally provide predicates and functions on time attributes, which may be applied to, for example, valid time, and queries such as these arise naturally. Enlarging the applicability of the SEQ VT modifier to statements that include predicates and functions on valid and transaction time offers a higher degree of orthogonality and wider ranging temporal support.

Intuitively, extended S-reducibility requires that the presence of explicit time references does not change the S-reducible evaluation mode of the rest of the statement (cf., Example 2.5). Let $tnorm(q^t)$ be the query where each explicit time reference in q^t is replaced by the unit period PERIOD '1-1'. Let $\langle\langle q^t(db^t)\rangle\rangle_{M^t}^{tnorm}$ be the result of evaluating q^t on db^t in model M^t, with the addition that all explicit time references evaluate to the unit period.

Definition 2.6 (*Extended S-reducibility*). Let $M^s = (DS^s, QL^s)$ be a nontemporal data model, and $M^t = (DS^t, QL^t)$ be an S-reducible extension of M^s. Model M^t is an *extended S-reducible* extension of M^s iff for any S-reducible query q^t in QL^t that

$$\forall q_i^t \in QL^t(tnorm(q_i^t) = q^t \Rightarrow \langle\langle tnorm(q_i^t)\rangle\rangle_{M^t} = \langle\langle q_i^t\rangle\rangle_{M^t}^{tnorm}).$$

Extended S-reducibility enlarges the scope of S-reducibility to statements that cannot be answered by considering the snapshots of a temporal

database in isolation. This is an important and nontrivial extension. As a direct consequence, it is no longer possible to define the semantics purely in terms of individual snapshots. Instead, the semantics of such statements must be defined explicitly. Statements that may include functions and predicates on timestamps are defined in Section 4.

2.6 Interval Preservation

Coupling snapshot reducibility (Definition 2.4) with syntactical similarity (Definition 2.5), and using this property as a guideline to semantically and syntactically embed temporal functionality in a language, is attractive but also limited. S-reducibility and extended S-reducibility adopt a point-based view, and see the intervals associated with data only as convenient representations of time points, supporting a point-based view. This view is pervasive in database research [Tansel et al. 1993]. In contrast, the *interval-based view* is quite pervasive in artificial intelligence research [Allen 1983; van Benthem 1991]. Here, intervals are not merely containers of points, but are atomic values that cannot be split or merged without changing the meaning of the data. In particular, S-reducibility is point-based: it does not distinguish between distinct relations if they contain the same snapshots, that is, if they are snapshot-equivalent [Jensen et al. 1994]. This means that many separate results of an S-reducible query are generally possible: the results will be snapshot-equivalent, but will differ in how the resulting tuples are timestamped. As a simple example, if $\{\langle X \| 1-5\rangle\}$ is a possible result of an S-reducible query, so is $\{\langle X \| 1-2\rangle, \langle X \| 3-5\rangle\}$. In this section we delve into the issue of which result(s) should be favored out of the many possible permitted by S-reducibility.

To illustrate in more detail, consider the following two relations, which are different and may be given different meanings by the user. Employees are evaluated every June and December, and bonuses are allocated for the past six months. Bob's performance was judged satisfactory, and so his bonuses were set at 20 for the first and last halves of the year, as shown in the left relation.

Bonus		
ID	Name	VTIME
1	20	1–6
1	20	7–12

Bonus 2		
ID	Amt	VTIME
1	20	1–12

Now consider the relation to the right. In spite of the difference between relations, they are the same in the snapshot reducibility context because a relation is viewed as no more than a collection of time-indexed nontemporal relations: the two relations imply exactly the same nontemporal relations. We thus have an example where two distinct relations—with quite different meanings in terms of what is important for the application user—cannot be differentiated by snapshot-equivalence.

The difference between the two relations is that one is coalesced while the other is not [Böhlen et al. 1996]. 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, and some queries evaluated on a coalesced temporal relation may produce an uncoalesced result [Böhlen et al. 1996].

When formulating more specific design desiderata for timestamping the tuples of query results, two possibilities come to mind. One possibility is to require the 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 not preserved and that with more than one interval-valued time dimension, no unique coalesced relation exists (as demonstrated in 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 let statement modifiers accommodate both possibilities in the same language. The default is to preserve the timestamps—since it is irreversible, coalescing cannot be the default.

In the sequel we define interval preservation [Böhlen et al. 1998a], which intuitively requires that timestamps be changed as little as possible. We assume valid time relations throughout, and use the following auxiliary notions. Temporal relations s_1 and s_2 are *snapshot-equivalent*, $s_1 \stackrel{se}{=} s_2$, iff at each point in time their snapshots are identical [Jensen et al. 1994]. Two tuples are *value-equivalent* iff their explicit attributes are pairwise-identical [Böhlen et al. 1996]. Finally, a *temporal element* is a finite union of intervals [Gadia 1986].

As first steps, we define three auxiliary notions, namely *output points*, *required argument tuples*, and *maximal impact tuples*.

Example 2.6 The following database and query are used to illustrate the definitions.

- Schemas: $R1(A \| VT)$, $R2(B, C \| VT)$, and $R(D \| VT)$.
- `r1` = $\{\langle 1 \| 5-8 \rangle\}, \langle 3 \| 1-3 \rangle, \langle 3 \| 4-12 \rangle, \langle 4 \| 1-5 \rangle$
- `r2` = $\{\langle 1, 2 \| 4-10 \rangle, \langle 3, 2 \| 6-9 \rangle, \langle 4, 2 \| 6-9 \rangle\}$
- `Q = SEQ VT SELECT A FROM r1 WHERE NOT EXISTS (SELECT * FROM r2 WHERE B > A)`
- `r` = $\{\langle 1 \| 5-5 \rangle, \langle 3 \| 1-3 \rangle, \langle 3 \| 4-5 \rangle, \langle 3 \| 10-12 \rangle, \langle 4 \| 1-5 \rangle\}$

Definition 2.7 (*Output points, op*). Let Q be a query, $r_1, \ldots, r_n$ be relations, and $\mathbf{x}$ be a sequence of explicit attribute values. The *output points* is a formula

$$op(Q, r_1, \ldots, r_n, \mathbf{x}, A)$$

where

(1) $r_1, \ldots, r_n$ are the argument tuples of query Q;,

(2) A is the set of time points associated with the result tuple $\mathbf{x}$; and

(3) $op(Q, r_1, \ldots, r_n, \mathbf{x}, A) \wedge op(Q, r_1, \ldots, r_n, \mathbf{x}, A') \Rightarrow A = A'$.

We use the output points to define the set of time points that the result of a temporal query will include. The output points have to be defined for each query or class of queries individually. Assume the S-reducible query Q = SEQ VT Q' from our running example. For the family of S-reducible queries, the set of output points is defined in terms of the individual snapshots of the database:

$$op(Q, r_1, \ldots, r_n, \mathbf{x}, A) \text{ iff } \forall t(\mathbf{x} \in Q'(\tau_t^{vt}(r_1), \ldots, \tau_t^{vt}(r_n)) \Leftrightarrow t \in A).$$

The valid-time timeslice operator, τ_t^{vt}, takes as arguments a temporal relation r and a time instant t and returns a relation r' without a valid-time dimension. The relation r' contains all tuples that are valid at time t, that is, those tuples of r whose associated valid time includes the time point t, but without the valid time. The formal definitions can be found in Appendix B.

Example 2.7 With Q and DB from our example, we get the following sets of output points:

- op(Q, r1, r2, $\langle 1 \rangle$, $\{5\}$)
- op(Q, r1, r2, $\langle 3 \rangle$, $\{1, 2, 3, 4, 5, 10, 11, 12\}$)
- op(Q, r1, r2, $\langle 4 \rangle$, $\{1, 2, 3, 4, 5\}$)

The next step is to characterize the set of *required argument tuples* for each potential result tuple $\langle \mathbf{x} \| I \rangle$. The required argument tuples, $s_1, \ldots, s_n$, are a minimal subset of the argument tuples, such that the output points still include all the time points of I.

Definition 2.8 (*Required argument tuples, rat*). Let $r_1, \ldots, r_n$ be the argument tuples, Q be a query, $\mathbf{x}$ be a sequence of attribute values, and I be an interval. A subset of the argument tuples, $s_1, \ldots, s_n$, is *required* for the result tuple $\langle \mathbf{x} \| I \rangle$ iff $\mathbf{x}$ and I can be entirely determined from $s_1, \ldots, s_n$, but not from any proper subset of $s_1, \ldots, s_n$.

$rat(r_1, \ldots, r_n, Q, \mathbf{x}, I, s_1, \ldots, s_n)$ iff

(1) $op(r_1, \ldots, r_n, Q, \mathbf{x}, A) \wedge I \subseteq A \wedge$

(2) $s_1 \subseteq r_1 \wedge \ldots \wedge s_n \subseteq r_n \wedge op(s_1, \ldots, s_n, Q, \mathbf{x}, A') \wedge I \subseteq A' \wedge A' \subseteq A \wedge$

(3) $\forall s'_1, \ldots, s'_n(s'_1 \subseteq s_1 \wedge \ldots \wedge s'_n \subseteq s_n \wedge \bigcup s'_i \subset \bigcup s_i \wedge op(s'_1, \ldots, s'_n, Q, \mathbf{x}, A'') \Rightarrow (I \not\subseteq A'' \vee A'' \not\subseteq A'))$

Line (1) states that only result intervals that are subsets of the set of output points are considered. (An interval I is a subset of a temporal element A iff each time point in I is also in A: $I \subseteq A$ iff $\forall t(t \in I \Rightarrow t \in A)$.) Line (2) requires that the output points of the required argument tuples still include all points of the result interval ($I \subseteq A'$) and that no new points have been introduced ($A' \subseteq A$). Finally, line (3) ensures that the set of required argument tuples is minimal for the given result tuple.

Example 2.8 With `r1`, `r2`, and `Q` from our example, the following holds true.

- rat(`r1`, `r2`, `Q`, $\langle 3 \rangle$, $1-3$, $\{\langle 3 \| 1-3 \rangle\}$, $\{\}$)
- rat(`r1`, `r2`, `Q`, $\langle 3 \rangle$, $1-2$, $\{\langle 3 \| 1-3 \rangle\}$, $\{\}$)
- rat(`r1`, `r2`, `Q`, $\langle 3 \rangle$, $4-5$, $\{\langle 3 \| 4-12 \rangle\}$, $\{\langle 4, 2 \| 6-9 \rangle\}$)
- rat(`r1`, `r2`, `Q`, $\langle 3 \rangle$, $1-5$, $\{\langle 3 \| 1-3 \rangle, \langle 3 \| 4-12 \rangle\}$, $\{\langle 4, 2 \| 6-9 \rangle\}$)
- rat(`r1`, `r2`, `Q`, $\langle 3 \rangle$, $10-12$, $\{\langle 3 \| 4-12 \rangle\}$), $\{\langle 4, 2 \| 6-9 \rangle\}$

Not all rat-relationships are equally interesting. First, some relationships do not maximize the output interval I. (The first and second relationships illustrate this.) The reason is that the required argument tuples are defined for all possible subintervals of a temporal element. For example, assume a scenario with exactly one required argument tuple: $rat(r, Q, \mathbf{x}, I, \{t\})$. In this case, $rat(r, Q, \mathbf{x}, I', \{t\})$ also holds for all subintervals $I' \subseteq I$.

A second property of the required argument tuples is that, for long result intervals, it is usually necessary to combine multiple argument tuples. (The fourth relationship illustrates this.) It combines the first and third rat-relationship into a single relationship.

We use the set *maximal impact tuples* to characterize the required argument tuples of interest to us.

Definition 2.9 (*Maximal impact tuples, mit*). Let $r_1, \ldots, r_n$ be argument relations, Q be a query, $\mathbf{x}$ be a sequence of attribute values, I be a time interval, and $s_1, \ldots, s_n$ be required argument tuples. $\langle \mathbf{x} \| I \rangle$ is a *maximal impact tuple* if I can neither be enlarged nor reduced, that is,

$mit(r_1, \ldots, r_n, Q, \mathbf{x}, I, s_1, \ldots, s_n)$ iff

(1) $rat(r_1, \ldots, r_n, Q, \mathbf{x}, I, s_1, \ldots, s_n) \wedge$

(2) $\neg \exists I'(I' \supset I \wedge rat(r_1, \ldots, r_n, Q, \mathbf{x}, I', s_1, \ldots, s_n) \wedge$

(3) $\neg \exists I'', s'_1, \ldots, s'_n(I'' \subset I \wedge s'_1 \subseteq r_1 \wedge \ldots \wedge s'_n \subseteq r_n \wedge \bigcup s'_i \subset \bigcup r_i \wedge rat(r_1, \ldots, r_n, Q, \mathbf{x}, I'', s'_1, \ldots, s'_n))$

The relationship defined by the maximal impact tuples is a subset of the relationship defined by the required argument tuples (line (1)). Line (2) excludes required argument tuples that permit a larger result interval. This ensures that the result tuple $\langle \mathbf{x} \| I \rangle$ is maximal for a given set of required argument tuples. Line (3) excludes result tuples that are constructed by combining independent required argument tuples.

Example 2.9 With r1, r2, Q, and r from our example, the following holds true.

- mit(r1, r2, Q, $\langle 1 \rangle$, $5-5$, $\{\langle 1 \| 5-8 \rangle\}$), $\{\langle 3, 2 \| 6-9 \rangle\}$
- mit(r1, r2, Q, $\langle 3 \rangle$, $1-3$, $\{\langle 3 \| 1-3 \rangle\}$, $\{\}$)
- mit(r1, r2, Q, $\langle 3 \rangle$, $4-5$, $\{\langle 3 \| 4-12 \rangle\}$), $\{\langle 4, 2 \| 6-9 \rangle\}$
- mit(r1, r2, Q, $\langle 3 \rangle$, $10-12$, $\{\langle 3 \| 4-12 \rangle\}$), $\{\langle 4, 2 \| 6-9 \rangle\}$
- mit(r1, r2, Q, $\langle 4 \rangle$, $1-5$, $\{\langle 4 \| 1-5 \rangle\}$, $\{\}$).

Using the concepts developed so far, we define the notion of interval preservation.

Definition 2.10 (*Interval preservation*). A query Q is *interval-preserving* iff for all argument tuples, $r_1, \ldots, r_n$, the query result, $Q(r_1, \ldots, r_n)$, coincides with the maximal impact tuples:

$$\forall r_1, \ldots, r_n, \mathbf{x}, I(\langle \mathrm{x} \| I \rangle \in Q(r_1, \ldots, r_n) \Leftrightarrow \exists s_1, \ldots, s_n(mit(r_1, \ldots, r_n, Q, \mathbf{x}, I, s_1, \ldots, s_n))).$$

Example 2.10 In our example, the set of maximal impact tuples is $\phi = \{\langle 1 \| 5-5 \rangle\}, \langle 3 \| 1-3 \rangle, \langle 3 \| 4-5 \rangle, \langle 3 \| 10-12 \rangle, \langle 4 \| 1-5 \rangle$ (cf., Example 2.9).

The result r $= \{\langle 1 \| 5-5 \rangle, \langle 3 \| 1-3 \rangle, \langle 3 \| 4-5 \rangle, \langle 3 \| 10-12 \rangle, \langle 4 \| 1-5 \rangle\}$ is consistent with Q being interval-preserving because $r = \phi$.

On the other hand, consider r$' = \{\langle 1 \| 5-5 \rangle, \langle 3 \| 1-5 \rangle, \langle 3 \| 10-12 \rangle, \langle 4 \| 1-5 \rangle\}$, which results from coalescing r. With this result Q is not interval-preserving because r$' \neq \phi$.

Finally, consider r$'' = \{\langle 1 \| 5-5 \rangle, \langle 3 \| 1-3 \rangle, \langle 3 \| 4-5 \rangle, \langle 3 \| 10-12 \rangle, \langle 4 \| 1-2 \rangle, \langle 4 \| 3-5 \rangle\}$, where an interval in r has been split. Again, r$'' \neq \phi$. Thus, Q is not interval-preserving.

2.7 Nonrestrictiveness

Sequenced statements are attractive because they provide specific built-in temporal semantics based on viewing a database as a sequence of states. However, some queries may need different semantics and cannot be expressed as sequenced queries. We would like to ensure that querying is not necessarily constrained by sequenced semantics.

Definition 2.11 (*Nonrestrictiveness*). A temporal query language is *nonrestrictive* iff it offers statements that manipulate timestamps as regular (interval) values, with no built-in temporal semantics enforced, and vice versa.

Nonrestrictiveness guarantees the availability of statements with the standard nontemporal semantics. This is particularly important in the context of migration, where users are expected to be well acquainted with the semantics of their nontemporal language. The requirement ensures that users are able to keep using the paradigm they are familiar with and to incrementally adopt new features. Moreover, from a theoretical perspective, any variant of temporal logic—a well-understood language that only provides built-in temporal semantics—is strictly less expressive than a language in first-order logic with explicit references to time, that is, a nonrestrictive language [Toman and Niwiński 1996].

Example 2.11 We use the modifier `NSEQ VT` to signal standard SQL semantics with full explicit control over timestamp attributes, and we term the resulting statements "nonsequenced" (this choice of this term is discussed in the next section). Finally, we use the modifier `SET VT` *range* to convert a nontemporal table with interval data into a temporal table.

```
NSEQ VT
  SELECT E.ID FROM Employee AS E, Salary AS S
  WHERE VTIME(E) PRECEDES VTIME(S) AND E.ID = S.ID;

CREATE TABLE Employee2 (ID INTEGER, Per PERIOD);

CREATE VIEW TEmployee2 AS SET VT Per SELECT ID FROM Employee2;
```

The query joins `Employee` and `Salary`. The join is not performed at each snapshot. Instead, we require that the valid time of `Employee` precedes the valid time of `Salary`. The result is a nontemporal table. Table `Employee2` is a nontemporal table with an explicit period column. `TEmployee2`, on the other hand, is a temporal view of this table, interpreting the `Per` column as the implicit timestamp. A *range specification* (`SET VT`, cf., Section 3.2.3) is used to explicitly set the valid time of `TEmployee2`.

This last view is useful in the situation introduced in Section 2.3, in which the application stores time-varying information in conventional relations by using date or time attributes. Application programmers can define temporal views, effectively converting such relations into temporal relations in order to utilize temporal semantics. As timestamps are encoded in nontemporal tables in a great variety of ways, a comprehensive approach to providing temporal support to legacy applications managing time-varying data cannot be solved entirely via query language design, but requires tools that help the application developer define appropriate temporal views and simplify legacy query statements by referencing these views instead.

Unlike S-reducible statements, nonsequenced statements do not offer built-in support, but offer complete control instead (this is akin to programming in assembly language, where one can do everything, but everything is hard to do). The query language must provide a set of functions and predicates for expressing temporal relationships (e.g., `PRECEDES`) and for performing manipulations and computations on timestamps (e.g., `VTIME`). The resulting new query-language constructs are relatively easy to integrate because they only require changes at the level of built-in predicates and functions.

3. APPLYING STATEMENT MODIFIERS TO SQL-BASED LANGUAGES

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

We now describe the constrained language design space and demonstrate the practical utility of statement modifiers for meeting the desiderata. Specifically, this section develops a design for ATSQL, an SQL-based temporal language. We chose SQL-92 as the concrete context because it is a rather complex language and is in widespread use. However, statement modifiers are not restricted to a specific language.

3.1 Global Impact on the Design Space

Upward compatibility dictates that the temporal language contain 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 duration, and another keyword must be chosen for the interval data type—we chose `PERIOD`. As another implication, the temporal extension must contend with *all* the facilities of SQL-92, for example, nested queries, aggregates, and null values. Finally, 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).

In order to satisfy temporal upward compatibility, all SQL-92 statements must work on temporal relations as well as on nontemporal 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. Queries, views, and constraints consider only the snapshot states of the argument, that is, temporal relations that are current and valid at the times they are evaluated.

The fact that syntactically similar snapshot-reducible temporal counterparts of all SQL-92 queries exist also affects the design. For each SQL-92 query, we must be able to pre- or append a fixed text string, that is, a *modifier*, to get the corresponding temporal query. We chose `SEQ VT` for the valid time and `SEQ TT` for transaction time, emphasizing that a temporal database is viewed as a sequence of nontemporal databases.

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, manipulating the timestamp. We chose the flags `NSEQ VT` and `NSEQ TT` for such queries. 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 queries as regular, explicit attributes.

To increase the utility of statement modifiers, we extend them with so-called *domain specifications*, making it possible to restrict the parts of

the argument tuples in queries to certain time periods. We also add *range specifications* that allow the specification of the timestamps of the resulting tuples.

3.2 Adding Detail to the Design

The desiderata 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 practice (e.g., generality and orthogonality) guides the design.

3.2.1 *Extensions at the Statement Level.* A first question is how to associate modifiers with the different kinds of statements in SQL, that is, with query expressions, views, assertions, integrity constraints, and modification statements. Section 2 simply requires that a statement modifier be placed at the beginning or end of a statement and that it apply to the statement as a whole. Within these restrictions, there are several possibilities for positioning the statement modifiers for the different types of statements.

We provide an EBNF syntax for each extension of SQL-92. We focus on the temporal extensions, and omit some details of SQL-92. In the EBNF productions, terminals take the form "xxx", that is, enclosed in quotation marks. Nonterminals of the form derive from the SQL-92 standard [Melton and Simon 1993], and new nonterminals are of the form <xxx> . Omitting these nonterminals yields the original SQL-92 productions.

Table definitions are extended to permit declaration of valid-time, transaction-time, and bitemporal tables.

```
<table definition> ::= "CREATE" "TABLE" <table name><table element list>
                       ["AS" "VT" [ "AND" "TT"] ] [ "AS" " TT" ]
```

In queries, view definitions, and declare cursor statements, the statement modifiers are placed at the outermost level, outside the query expression. We specify this by augmenting the definition of the nonterminal.

```
<query expression'>::=
  <modifiers><query expression> |
  "(" <modifiers><query expression> ")" <coal>
```

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 nonterminal <coal> is used for specifying coalescing (to be discussed later in this section). Statement modifiers can also be associated with query expressions in derived table expressions in from clauses. The motivation is that derived tables may be meaningfully computed independently 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 allowing derived tables expressions to have their own individual statement modifiers.

Note that 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, separate modifiers are not allowed for subqueries.

In assertions, statement modifiers are placed in front of the assertion by augmenting the <assertion check> nonterminal.

```
<assertion check'> ::= <modifiers> <assertion check>
```

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

```
<column constraint'> ::= <modifiers> <column constraint>

<table constraint'> ::= <modifiers> <table constraint>
```

Finally, 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>
```

3.2.2 *Statement Modifiers.* We start with an EBNF syntax for statement modifiers, and then discuss their meanings.

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

The meaning of a statement modifier 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 discuss the core semantics in this section, deferring domain and range specifications and coalescing to the next sections.

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

No modifier. A missing modifier for a time dimension (i.e., valid time or transaction time) dictates upward compatibility (UC) when none of the underlying argument relations support that time; otherwise, evaluation is dictated according to temporal upward compatibility (TUC). The time dimension will not be present in the result of a query.

SEQ. When this keyword is present for a time dimension, evaluation consistent with sequenced semantics (SEQ), that is, built-in timestamp-related processing, is dictated for that time dimension. The time dimension will be present in query results.

NSEQ. This keyword signals nonsequenced semantics (NSEQ), that is, timestamp processing with no built-in semantics enforced by the DBMS. The affected time dimension is not present in query results (with this

Table II. Basic Usage of Statement Modifiers

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 VT <SQL-92 >	(T)UC	SEQ
NSEQ TT <SQL-92 >	(T)UC	NSEQ
SEQ TT AND SEQ TT < SQL-92 >	SEQ	SEQ
SEQ TT AND NSEQ TT < SQL-92 >	SEQ	NSEQ
NSEQ TT AND SEQ TT < SQL-92 >	NSEQ	SEQ
NSEQ TT AND NSEQ TT < SQL-92 >	NSEQ	NSEQ

modifier, the time becomes an explicit attribute and can be included in the result, similarly to the inclusion of other explicit attributes).

With two time dimensions, the three cases lead to a total of nine kinds of statements, as summarized in Table II. The different orderings of the valid- and transaction-time modifiers are omitted, as they have the same semantics.

3.2.3 *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. It restricts the (argument) database to the part that is valid or current during that period. A domain restriction is applied prior to the evaluation of a statement, that is, in a preprocessing step.

Example 3.1 Consider the following two statements.

```
SEQ VT PERIOD '1996-1999'
   SELECT E.Name, S.Amt FROM Employee AS E, Salary AS S WHERE E.ID = S.ID;

CREATE TABLE Employee3 (ID INTEGER, SEQ VT PERIOD '10-20' PRIMARY KEY(ID));
```

The domain restriction in the query says that we are only interested in facts valid between 1996 and 1999. 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, that is, the *time range*. The SET VT clause is used for this purpose. Transaction time semantics forbids this kind of user interaction [Snodgrass and Ahn 1985]. The time range is set in a postprocessing step, that is, after the evaluation of a query.

Example 3.2

```
NSEQ VT SET VT PERIOD(BEGIN(VTIME(E)),END(VTIME(S)))
   SELECT E.ID, S.Amt FROM Employee AS E, Salary AS S
   WHERE VTIME(E) PRECEDES VTIME(S);
```

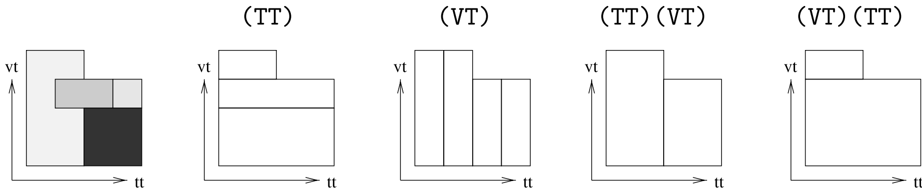

Fig. 1. Different forms of coalescing.

The statement joins `Employee`- and `Salary`-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.

3.2.4 *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 levels where modifiers are 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> ")"
```

Example 3.3 In the first statement below, we coalesce the results of a sequenced query. In the second query, we coalesce the relation in the from clause because we want the condition in the where clause to be evaluated over maximal valid times.

```
(SEQ VT SELECT * FROM Employee)(VT);

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

The semantics of coalescing depends on the type of relation it is applied to. A nontemporal 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 [Böhlen et al. 1996]).

The semantics of coalescing for bitemporal relations is more subtle. Here, overlapping or adjacent time regions (rectangles) of value-equivalent tuples must be merged. In the general case, overlapping rectangles do not coalesce into a single rectangle, which means that several result tuples must 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 1 exemplifies bitemporal coalescing. The first segment of Figure 1 displays the rectangular shapes defined by the timestamps of four value-equivalent tuples. The second and third segments

illustrate the two basic coalescing operations; coalescing in transaction time and coalescing in valid time. These two basic operations can be combined into `(TT)(VT)`, which means that we first coalesce in transaction time and then in valid time. As exemplified by the last two segments, the sequence of coalescing operations matters. Sequence `(TT)(VT)` results in maximal valid-time periods, whereas `(VT)(TT)` results in maximal transaction-time periods.

4. A FORMAL SEMANTICS

The previous section showed how statement modifiers can be used to generalize a nontemporal query language to one that includes temporal support, in a manner that satisfies the desiderata stated in Section 2. While this approach was exemplified in SQL-92, it is amenable to any nontemporal query language.

Just as the desiderata imply specific syntactic extensions, they also imply a specific semantics for those extensions, and interestingly also imply a way to extend an existing implementation of the original nontemporal query language to support the augmented temporal language. In this way, we can start with a nontemporal query language and implementation, and thus derive a temporal language semiautomatically, including its semantics and implementation.

As the details depend on the underlying language, we examine the augmentation process using the temporal extension of SQL-92 presented in previous sections. The temporal semantics $[\![\langle construct \rangle]\!]_{temp}$ is expressed in SQL-92 semantics $[\![\langle construct \rangle]\!]_{SQL-92}$. In a similar way, the semantics for a temporal extension of language X, $[\![\langle X \rangle]\!]_{temp}$, can be expressed in the underlying semantics, $[\![\langle X \rangle]\!]_X$.

In the case of SQL-92, we express its semantics in the standard relational algebra. For current and nonsequenced queries, we utilize new algebraic operators (τ^{vt}, τ^{tt}, SN^{vt}, SN^{tt}) that convert temporal relations to conventional relations. For sequenced queries, we generalize each relational algebra operator $\mathcal{O}$ to three sequenced variants, $\mathcal{O}^{vt}$, $\mathcal{O}^{tt}$, and $\mathcal{O}^{bi}$, that effect the appropriate semantics. We could apply the same approach to any procedural implementation of a nontemporal query language to arrive at an implementation of a temporal extension of that language, demonstrating that statement modifiers are amenable to 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. The first step is the translation of constructs at the level of functions and predicates. This step is straightforward, and is discussed in the first section. The second step is the translation at the statement level, that is, the translation of statements enhanced with statement modifiers, which is much more involved (and important!) and is covered in the subsequent three sections.

4.1.1 *Constructs for Timestamp Manipulation.* Temporal query languages generally define a variety of constructs to manipulate their various timestamp types, including 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 [Snodgrass 1995]. They are relatively easy to define, and adding one more construct to a language has only a localized effect on language design. Hence we only define a relatively small number of constructs here.

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

4.1.2 *Query Expressions.* We define the meaning of temporal query expressions by translating them into well-defined algebraic expressions. As a precursor, we introduce the notation used 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 nontemporal, valid-time, transaction-time, and bitemporal relations, respectively. The vertical double bar "$\|$" separates the explicit attributes from the implicit timestamps, and VT and TT denote valid time and transaction time, respectively.

In the definitions, we need auxiliary operators that time-slice relations and turn timestamps into regular, explicit attributes. These operators, defined formally in Appendix B, are overloaded to apply to valid-time, transaction-time, and bitemporal relations, and they have variants for both valid and transaction times. There are two time-slice operations. The first, τ_{tp}, selects all tuples in the argument relation with a timestamp that overlaps time point tp. The time dimension used in this selection is omitted in the result relation. The second timeslice operation, δ_{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. With these conventions in place, Table III gives the semantics for core statements (cf., Table II).

In the table, $[\![\langle \text{SQL-92} \rangle]\!]_{SQL\text{-}92}$ evaluates to the standard relational algebra expression that corresponds to $\langle$SQL-92$\rangle$ [Ceri and Gottlob 1985; Van Gelder and Topor 1991]. $[\![\langle \text{SQL-92} \rangle]\!]_T$, where $T \in \{vt, tt, bi\}$ evaluates to the same algebraic expression as $[[\langle \text{SQL-92} \rangle]]_{SQL\text{-}92}$, except that every nontemporal relational algebra operator (e.g., $\times$, σ, π) is replaced by the corresponding temporal relational algebra operator (e.g., $\times^{vt}$, σ^{vt}, π^{vt}). The

Table III. Core Semantics

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

$$\llbracket \underline{\texttt{SEQ VT}}<\text{SQL-92}> \rrbracket_{temp}(r_1, \ldots, r_n) \triangleq \llbracket <\text{SQL-92}> \rrbracket_{vt}(\tau^{tt}_{now}(r_1), \ldots, \tau^{tt}_{now}(r_n))$$

$$\llbracket \underline{\texttt{NSEQ VT}}<\text{SQL-92}> \rrbracket_{temp}(r_1, \ldots, r_n) \triangleq \llbracket <\text{SQL-92}> \rrbracket_{SQL\text{-}92}(\tau^{tt}_{now}(SN^{vt}(r_1)), \ldots, \tau^{tt}_{now}(SN^{vt}(r_n)))$$

$$\llbracket \underline{\texttt{SEQ TT}}<\text{SQL-92}> \rrbracket_{temp}(r_1, \ldots, r_n) \triangleq \llbracket <\text{SQL-92}> \rrbracket_{tt}(\tau^{vt}_{now}(r_1), \ldots, \tau^{vt}_{now}(r_n))$$

$$\llbracket \underline{\texttt{NSEQ TT}}<\text{SQL-92}> \rrbracket_{temp}(r_1, \ldots, r_n) \triangleq \llbracket <\text{SQL-92}> \rrbracket_{SQL\text{-}92}(\tau^{vt}_{now}(SN^{tt}(r_1)), \ldots, \tau^{vt}_{now}(SN^{tt}(r_n)))$$

$$\llbracket \underline{\texttt{SEQ VT AND SEQ TT}}<\text{SQL-92}> \rrbracket_{temp}(r_1, \ldots, r_n) \triangleq \llbracket <\text{SQL-92}> \rrbracket_{bi}(r_1, \ldots, r_n)$$

$$\llbracket \underline{\texttt{SEQ VT AND NSEQ TT}}<\text{SQL-92}> \rrbracket_{temp}(r_1, \ldots, r_n) \triangleq \llbracket <\text{SQL-92}> \rrbracket_{vt}(SN^{tt}(r_1), \ldots, SN^{tt}(r_n))$$

$$\llbracket \underline{\texttt{NSEQ VT AND SEQ TT}}<\text{SQL-92}> \rrbracket_{temp}(r_1, \ldots, r_n) \triangleq \llbracket <\text{SQL-92}> \rrbracket_{tt}(SN^{vt}(r_1), \ldots, SN^{vt}(r_n))$$

$$\llbracket \underline{\texttt{NSEQ VT AND NSEQ TT}}<\text{SQL-92}> \rrbracket_{temp}(r_1, \ldots, r_n) \triangleq \llbracket <\text{SQL-92}> \rrbracket_{SQL\text{-}92}(SN^{tt}(SN^{vt}(r_1)), \ldots, SN^{tt}(SN^{vt}(r_n)))$$

algebras are defined in Section 4.2. Due to space limitations, we only consider queries (i.e., the select statement), and then only those statements that can be mapped into the five basic algebra operators. This approach can be generalized to other operators, such as aggregation, and to other types of statements, such as modifications statements, view definitions, and assertions and integrity constraints.

The following three examples illustrate mapping from the temporal extension to the augmented algebra. The argument relations are assumed to be bitemporal.

Example 4.1 We start with a current query, termed Q_1.

```
SELECT E.Name, S.Amt
FROM Employee AS E, Salary AS S
WHERE E.ID = S.ID
```

This query is defined by the relational algebra expression

$$\llbracket Q_1 \rrbracket_{temp}(\texttt{Employee, Salary}) = \pi_{E.Name,\, S.Amt}(\sigma_{E.ID=S.ID}(\tau^{tt}_{now}(\tau^{vt}_{now}(\texttt{Employee}/E)) \times \tau^{tt}_{now}(\tau^{vt}_{now}(\texttt{Salary}/S))))$$

Note that the mapping from SQL-92 queries to relational algebra is still the same. The only change is at the innermost portion of the expression, where each bitemporal table is rendered nontemporal by time-slicing on valid and transaction times, thereby ensuring temporal upward compatibility.

The query below, termed Q_2, is an example of a nonsequenced query.

```
NSEQ VT
  SELECT E.ID, S.Amt
  FROM Employee AS E, Salary AS S
  WHERE E.ID = S.ID
  AND VTIME(E) PRECEDES VTIME(S)
```

This query is defined by the relational algebra expression,

$$\llbracket Q_2 \rrbracket_{temp}(\texttt{Employee, Salary}) = \pi_{E.ID,\, S.Amt}(\sigma_{E.ID=S.ID}(\sigma_{VTIME(E)\ PRECEDES\ VTIME(S)}(SN^{vt}(\tau^{tt}_{now}(\texttt{Employee}/E)) \times SN^{vt}(\tau^{tt}_{now}(\texttt{Salary}/S)))))$$

Table IV. Definition of Time-Domain Restrictions

$$[\![\underline{\text{<mode> VT <time-domain>}}\ q^t]\!]_{temp}(r_1, \ldots, r_n) \triangleq [\![\underline{\text{<mode> VT}}\ q^T]\!]_{temp}(\delta^{vt}_{\text{<time-domain>}}(r_1), \ldots, \delta^{vt}_{\text{<time-domain>}}(r_n))$$

$$[\![\underline{\text{<mode> TT <time-domain>}}\ q^t]\!]_{temp}(r_1, \ldots, r_n) \triangleq [\![\underline{\text{<mode> TT}}\ q^T]\!]_{temp}(\delta^{tt}_{\text{<time-domain>}}(r_1), \ldots, \delta^{tt}_{\text{<time-domain>}}(r_n))$$

The temporal predicate in the second selection can be viewed as a syntactic shorthand for a standard selection condition (cf. Table IX). The only additions are the "adjustments" of the relations (SN^{vt} and τ^{tt}_{now}) to the nonsequenced evaluation mode for valid time and the temporal upward-compatible evaluation mode for transaction time.

The following query, termed Q_3, is sequenced in both valid and transaction times.

```
SEQ VT AND SEQ TT
   SELECT E.Name, S.Amt
   FROM Employee AS E, Salary AS S
   WHERE E.ID = S.ID
```

It is defined by the temporal relational algebra expression below.

$$[\![Q_3]\!]_{temp}(\texttt{Employee, Salary}) = \pi^{bi}_{E.Name,\, S.Amt}(\sigma^{bi}_{E.ID=S.ID}(\texttt{Employee}/E \times^{bi} \texttt{Salary}/S))$$

Apart from the superscripts on the operators, the translation from SQL-92 queries to relational algebra expressions remains the standard one.

4.1.3 *Domain and Range Specifications*. A time-domain restriction constrains the argument relations in a query to contain only those tuples that are valid during a specific period, and only the parts of the tuples that intersect with the time-domain restriction are considered in the query. This is formalized in Table IV.

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 V.

4.1.4 *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)(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

Table V. Definition of Time-Range Specifications

$$[\![\underline{\texttt{SET VT}}\ range\ q^T]\!]_{temp}(r_1, \ldots, r_n) \triangleq$$

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

Table VI. Definition of Coalescing

$$[\![(q^T)\underline{(\texttt{VT})}]\!]_{temp}(r_1, \ldots r_n) \triangleq$$

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

$$[\![(q^T)\underline{(\texttt{TT})}]\!]_{temp}(r_1, \ldots r_n) \triangleq$$

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

query returns a bitemporal relation, it may be coalesced in valid time, in transaction time, or in a combination of the two. Table VI provides the definitions. A representative version of the algebraic coalescing operator, $coal$, will be defined shortly.

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.

Codd's relational algebra is given in Table VII, where c is a predicate and f is a generalized projection function that roughly corresponds to the select list of an SQL-92 statement.

The temporal relational algebra generalizes this nontemporal relational algebra, in two very specific senses: (1) each temporal operator must be snapshot-reducible to its nontemporal counterpart (cf., Section 2.4), even in the presence of constructs that explicitly reference the time (cf., Section 2.5); and (2) each temporal operator must be interval-preserving (cf.,

Table VII. The Nontemporal Relational Algebra

$\sigma_c(r)$	$\triangleq$	$\{t \mid t \in r \wedge c(t)\}$
$\sigma_c(r)$	$\triangleq$	$\{t \mid \exists t_2(t_2 \in r \wedge t_1 = f(t_2))\}$
$r_1 \cup r_2$	$\triangleq$	$\{t \mid t \in r_1 \vee t \in r_2\}$
$r_1 \times r_2$	$\triangleq$	$\{t_1 \circ t_2 \mid t_1 \in r_1 \wedge t_2 \in r_2\}$
$r_1 \backslash r_2$	$\triangleq$	$\{t \mid t \in r_1 \wedge t \notin r_2\}$

Section 2.6). Hence the desiderata in Table I significantly constrain the design even of the temporal relational algebra.

Figure 8 contains the definition of the valid-time version of the temporal relational algebra. The transaction-time version is omitted because it is similar to the valid-time version, 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 are defined in Table IX. Symbol "∘" denotes tuple concatenation.

The most complex operator is the 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 shown in Figure 2. The second line of the definition in Table VIII 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 an r_2 tuple ends. The third line then identifies all potential end points of periods of result tuples. The last line of the definition excludes "false" result tuples, by eliminating meaningless combinations of starting and ending points, as well as tuples with excessive periods.

Coalescing is the only operation without a nontemporal counterpart. It is also special because it destroys the representation of timestamps in order to enforce a particular representation (maximum periods). Coalescing merges overlapping or adjacent value-equivalent tuples, as illustrated in Figure 3 and defined next.

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

The two tuples introduced in the first line define the start (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^-, that is, there must exist another tuple with a valid time containing its start time. Finally, the last line ensures that the valid time of the result tuple is maximal, that is, there must not exist another tuple that contains either VT^- or VT^+.

Table VIII. Valid-Time Algebra

$\sigma_c^{vt}(r)$	$\triangleq$	$\{\langle t \Vert VT\rangle \mid \langle t \Vert VT\rangle \in r \wedge c(\langle t, VT\rangle)\}$
$\pi_f^{vt}(r)$	$\triangleq$	$\{\langle t \Vert VT\rangle \mid \exists t_2(\langle t \Vert VT\rangle \in r \wedge t_1 = f(\langle t_2, VT\rangle))\}$
$r_1 \cup^{vt} r_2$	$\triangleq$	$\{\langle t \Vert VT\rangle \mid \langle t \Vert VT\rangle \in r_1 \vee \langle t \Vert VT\rangle \in r_2\}$
$r_1 \times^{vt} r_2$	$\triangleq$	$\{\langle\langle t_1, VT_1\rangle \circ \langle t_2, VT_2\rangle \Vert VT\rangle \mid \langle t_1 \Vert VT_1\rangle \in r_1 \wedge \langle t_2 \Vert VT_2\rangle \in r_2 \wedge$ $VT = intersect(VT_1, VT_2) \wedge VT_1\ overlaps\ VT_2\}$
$r_1 \backslash^{vt} r_2$	$\triangleq$	$\{\langle t \Vert VT\rangle \mid \exists VT_1(\langle t \Vert VT_1\rangle \in r_1 \wedge$ $(\exists VT_2(\langle t \Vert VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = succ(VT_2^+)) \vee VT^- = VT_1^-) \wedge$ $(\exists VT_3(\langle t \Vert VT_3\rangle \in r_2 \wedge VT_1^+ \leq VT_3^- \wedge VT^+ = pred(VT_3^-)) \vee VT^+ = VT_1^+) \wedge$ $VT^- \leq VT^+ \wedge \neg\exists VT_4(\langle t \Vert VT_4\rangle \in r_2 \wedge VT_4\ overlaps\ VT))\}$

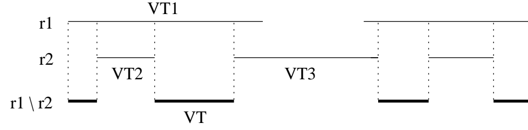

Fig. 2. Valid-time difference.

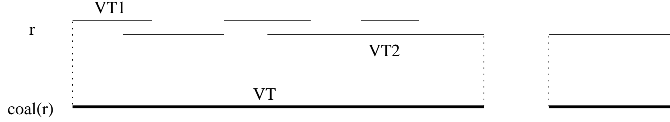

Fig. 3. Coalescing a valid-time relation.

While it is not possible to compute arbitrary transitive closures in SQL-92, coalescing is possible in SQL-92 because time is *linear* [Böhlen et al. 1996; Celko 1995].

The bitemporal relational algebra is a natural extension 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. This does not change the basic ideas, but does add to the complexity of the definitions, which can be found in Appendix C.

We note that care was taken to consider end points of valid and transaction timestamps only when defining the operators—intermediate time points are never used. This allows for a granularity-independent implementation, which is a precondition for an efficient execution of these operators.

5. PROPERTIES OF THE STATEMENT MODIFIER-EXTENDED SQL

We now evaluate the syntax and semantics of temporal statement modifiers applied to SQL-92. We argue that the extension does indeed satisfy the compatibility and reducibility properties introduced in Section 2. For brevity and to avoid tedious detail, we cover the valid-time dimension only.

- *Upward compatibility* with respect to SQL-92 is fulfilled by design. The approach adopted for defining the syntax and semantics emphasizes this property: The syntax is given by *extending* the syntax of SQL-92 with

nonmandatory 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 III covers SQL-92 statements, and ascribes to such statements the conventional SQL-92 semantics when applied to nontemporal relations.

- *Temporal upward compatibility* with respect to SQL-92 is ensured by the first definition in Table III. This definition includes SQL-92 statements when evaluated on temporal relations, and ascribes to such statements the conventional SQL-92 semantics on the current state. Note that care must also be taken to ensure TUC in the definition of SQL-92 modification statements (covered in detail elsewhere [Bair et al. 1997]) when applied to temporal relations.

- *S-reducibility* with respect to SQL-92 again follows from the definition of the language. Definition 2.5 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, respectively, to the SQL query. The fixed strings given in Section 3.2, specifically, <modifiers> do not depend on the particular query, but are the same for all queries. Hence, S-reducibility holds for a language design that carefully applies statement modifiers only before or after a conventional statement.

- *Extended S-reducibility* requires that S-reducibility apply even when sequenced queries include functions and predicates on timestamps (cf., Section 2.5). Functions on the timestamps can appear in the projection list and in the where clause; predicates on timestamps can appear in the where clause. We accommodate this in the temporal algebra in the π^{vt} and σ^{vt} operators (and their transaction time and bitemporal variants), as well as by ensuring that the timestamps are retained in the temporal Cartesian product operator. We must still demonstrate that these operators remain snapshot-reducible; we will turn to this question shortly.

- *Interval preservation* requires that timestamps are left unaltered, unless modification is required by the set of output points as, for example, defined by S-reducibility (cf., Section 2.6). For the relational model, this means that the query result coincides with the maximum impact tuples. This is proven below.

- *Nonrestrictiveness* requires that the timestamps be manipulable as regular values, and that conventional data be accessible with temporal semantics (cf., Section 2.7). NSEQ semantics, specifically the SN^{vt} and SN^{tt} functions in Table III, provide the former functionality; the SET VT semantics in Table V provides the latter, both utilize statement modifiers.

Hence, the extension using the statement modifier satisfies all the desiderata listed in Table I.

5.1 S-Reducibility

We now examine extended S-reducibility in more detail. Recall the definition of snapshot-reducibility (Definition 2.4). As the first step, we show that, with respect to relational algebra, valid-time relational algebra almost has this property.

THEOREM 5.1 *Valid-time relational algebra (Table VIII) satisfies the reducibility properties below with respect to relational algebra (Table VII).*

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

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 equivalences. Also, $\pi^{-}_{X}(r)$ *is given by* $\pi_{r.*\setminus X}(r)$. *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 nontemporal counterparts. Thus, all valid-time algebra statements involving only these operators are snapshot-reducible to the algebra statements obtained by simply removing the *vt* superscripts.

However, the equivalence involving the Cartesian product is not reducible to the nontemporal Cartesian product! While it is straightforward to define a snapshot-reducible temporal Cartesian product, we have chosen a definition that violates snapshot-reducibility. Let us explore the rationale for this design decision.

First, 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 π^{-} 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 when mapping SQL queries to relational algebra.

Example 5.1 Consider the following query.

```
SEQ VT
  SELECT p.X, VTIME(p)
  FROM p, q, r
  WHERE DURATION(VTIME(p),DAY) + DURATION(VTIME(q),DAY) < DURATION(VTIME(r),DAY)
```

We would like to map this query to

$$\pi^{vt}_{p.X,\, p.VT}(\sigma^{vt}_{DURATION(p.VT,\, DAY)+DURATION(q.VT,\, DAY)<DURATION(r.VT,\, DAY)}((\mathtt{p} \times^{vt} \mathtt{q}) \times^{vt} \mathtt{r})).$$

With our definition of the valid-time Cartesian product, the timestamps of the argument tuples are retained as explicit attributes, and they can be referenced in the projection.

Using a snapshot-reducible Cartesian product would make it impossible to construct a correct expression for the temporal projection operator, and the information required to evaluate the predicate would be lost. This observation holds for any timestamped tuple and any homogeneous attribute-value timestamped data model [Gadia 1988].

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. Note, though, that this returns results that are not valid-time relations, and thus breaks the closedness property of the algebra. This approach was adopted in the algebra for the HSQL data model [Sarda 1993], including both a reducible “Concurrent Product” and an information-preserving “Cartesian product.”

Another approach is to introduce an n-ary valid-time join that can then be defined to be snapshot-reducible [Soo et al. 1995].

Example 5.2 The transformation of the SQL in Example 5.1 to such a temporal algebra resulted in the following expression:

$$\pi^{vt}_{p.X,\, p.VT}(\bowtie^{vt}_{DURATION(p.VT,\, DAY)+DURATION(q.VT,\, DAY)<DURATION(r.VT,\, DAY)}(\mathtt{p},\, \mathtt{q},\, \mathtt{r}))$$

While the added complexity of an n-ary operator may be undesirable, there is still the problem of not having access to the valid timestamp of `p`. With the n-ary join approach, the original valid times of tuples cannot be inferred from the results of the join. With our Cartesian product, the needed values are readily available.

Most temporal algebras have operators that are snapshot-reducible with respect to the nontemporal Cartesian product (e.g., the TJOIN [Navathe and Ahmed 1989]; the Concurrent Product Operator [Sarda 1993]; the cross-product [Nair and Gadia 1993]; (temporal) equijoin [Clifford et al. 1993]; and the valid-time Cartesian product[3] [Snodgrass 1993]; McKenzie and Snodgrass [1991] give 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. Specifically, the nonreducibility of the operator does not lead to violations of S-reducibility to SQL. In SQL-based languages, Cartesian

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

products are specified using the FROM clause from the *query specifications* [Melton and Simon 1993]. 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 dimension of a relation; and defaults (e.g., SELECT *) do not expand to include the implicit time attributes. Thus, the additional explicit time attributes in the results of Cartesian products do not compromise the S-reducibility property due to the presence of subsequent projections.

5.2 Interval Preservation

THEOREM 5.2 *The valid-time relational algebra is interval-preserving, that is,*

- $\langle \mathbf{x} \| I \rangle \in \sigma_c^{vt}(r) \Leftrightarrow \exists s(mit(r, \sigma_c^{vt}, \mathbf{x}, I, s))$
- $\langle \mathbf{x} \| I \rangle \in \pi_f^{vt}(r) \Leftrightarrow \exists s(mit(r, \pi_f^{vt}, \mathbf{x}, I, s))$
- $\langle \mathbf{x} \| I \rangle \in (r_1 \cup^{vt} r_2) \Leftrightarrow \exists s_1, s_2(mit(r_1, r_2, \cup^{vt}, \mathbf{x}, I, s_1, s_2))$
- $\langle \mathbf{x} \| I \rangle \in (r_1 \times^{vt} r_2) \Leftrightarrow \exists s_1, s_2(mit(r_1, r_2, \times^{vt}, \mathbf{x}, I, s_1, s_2))$
- $\langle \mathbf{x} \| I \rangle \in (r_1 \setminus^{vt} r_2) \Leftrightarrow \exists s_1, s_2(mit(r_1, r_2, \setminus^{vt}, \mathbf{x}, I, s_1, s_2))$

PROOF. We start with valid-time selection. The output points are defined by the valid-time selection operator in Figure 8: $op(r, \sigma_c^{vt}, \mathbf{x}, A)$ iff $A = \bigcup_I(\langle \mathbf{x} \| I \rangle) \in r \wedge c(\langle \mathbf{x}, I \rangle)$In particular, if r contains exactly one tuple that satisfies condition c, then $A = I$. We show that a maximal impact tuple is given by a formula of the form $mit(r, \sigma_c^{vt}, \mathbf{x}, I, \{\langle \mathbf{x} \| I \rangle\})$, that is, there is exactly one argument tuple required, and the timestamp of the maximal impact tuple and the timestamp of the required argument tuple are identical.

- $mit(r, \sigma_c^{vt}, \mathbf{x}, I, \{\langle \mathbf{x} \| I' \rangle\}) \wedge I \subset I'$, that is, a situation where the timestamp of the maximal impact tuple is smaller than the timestamp of the required argument tuple, is impossible due to line (2) in Definition 2.9.
- $mit(r, \sigma_c^{vt}, \mathbf{x}, I, \{\langle \mathbf{x} \| I' \rangle\}) \wedge I \supset I'$, that is, a situation where the timestamp of the maximal impact tuple is larger than the timestamp of the required argument tuple, is impossible due to line (2) in Definition 2.8.
- $mit(r, \sigma_c^{vt}, \mathbf{x}, I, s) \wedge |s| > 1$, that is, a situation with more than one required argument tuple, is impossible due to line (4) in Definition 2.9 (which forbids combining argument tuples in order to yield a larger result interval); and line (3) of Definition 2.8 (which forbids argument tuples that do not uniquely contribute a point to the result interval).

Thus, each tuple that satisfies the selection condition is a result tuple of σ_c^{vt}, and is a maximal impact tuple. Therefore, $\langle \mathbf{x} \| I \rangle \in \sigma_c^{vt}(r) \Leftrightarrow \exists s(mit(r, \sigma_c^{vt}, \mathbf{x}, I, s))$, and intervals are preserved.

Projection and union exhibit the same properties, that is, there is always exactly one argument tuple required, and the timestamp of this tuple and the timestamp of the maximal impact tuple are identical. Therefore, both operations are interval-preserving as well.

The maximal impact tuples of the valid-time Cartesian product are given by the formula $mit(r_1, r_2, \times^{vt}, \mathbf{x}, I, \{\langle \mathbf{x_1} \| I_1 \rangle\}, \{\langle \mathbf{x_2} \| I_2 \rangle\}) \wedge I = I_1 \cap I_2$. The required argument tuples consist of exactly two tuples: one from the left and one from the right argument relation. The timestamps of these two tuples must overlap, and the timestamp of the result tuple is the intersection of the timestamps of the required argument tuples. It follows that the result tuples of $\times^{vt}$ and the maximal impact tuples coincide, and that the valid-time Cartesian product is interval-preserving.

The maximal impact tuples of the valid-time difference are given by a formula of the form, $mit(r_1, r_2, \backslash^{vt}, \mathbf{x}, I, \{\langle \mathbf{x} \| I' \rangle\}, \{\langle \mathbf{x} \| I_1 \rangle, \ldots, \langle \mathbf{x} \| I_n \rangle\}) \wedge I \in (I' - I_1) - \cdots - I_n$. The set of required argument tuples consist of exactly one tuple from the first argument relation and a set of value-equivalent tuples from the second relation, which overlap with the tuple from the first relation. Subtracting the interval timestamps of the tuples from the second relation from the interval timestamp of the first relation yields a set of intervals. These intervals are used to timestamp maximal impact tuples. Because the tuples from the first argument relation are considered individually and the intervals of these tuples are only split if required by the tuples in the second relation, valid-time difference is interval-preserving. □

6. RELATED WORK

This section covers related work on requirements, statement modifiers, and temporal SQL extensions.

6.1 Query Language 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 (e.g., Ariav [1986, p. 513]; Navathe and Ahmed [1993, p. 99]; Sarda [1993, pp. 123, 125]; Lorentzos and Mitsopoulos [1997, p.480]), we have found no precise definition of it.

We have encountered one temporal relational proposal that aims at satisfying a requirement that has some similarity with temporal upward compatibility. The TempSQL language (e.g., Gadia and Nair [1993]) 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 independently be made temporal, this would essentially (providing that a number of other design decisions are made correctly) yield a temporal upward-compatible SQL extension.

Formulating the notion of S-reducibility uses the fundamental notion of snapshot-reducibility [Snodgrass 1987], and was inspired by an informal concept presented in the context of the ChronoLog language. S-reducibility was formalized during the process of solving problems identified in TSQL2 [Böhlen et al. 1995], the goal was to develop a proposal for the SQL/Temporal part of SQL3 [Snodgrass et al. 1996a; 1996b; 1998].

The language desiderata beyond S-reducibility, that is, interval preservation, extended S-reducibility, and nonrestrictiveness, do not appear to have been studied by other authors. Theoretical aspects of interval preservation are the subject of Böhlen et al. [1998a].

6.2 Statement Modifiers

Temporal statement modifiers represent a new approach to adding temporal support to an existing language. A primitive form of statement modifier was used in ChronoLog [Böhlen 1994], a temporal extension of a Datalog-based language.

Earlier decendants of the work described in this article are the change proposals submitted to the SQL/Temporal part of SQL3 [Snodgrass et al. 1996a; 1996b; 1998] (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 conformant in the context of the standard and its associated peculiarities

6.2.1 *Existing Temporal Extensions of SQL*. To complete the coverage of related research, we evaluate existing temporal SQL proposals, including SQL-92, and relate their design to the statement modifier-based approach. We report compliance with desiderata if claimed in the proposal's documentation, or if noncompliance cannot be proved. We consider each SQL in turn and in chronological order. UC is satisfied by all proposals, and a more detailed study of TUC can be found elsewhere [Bair et al. 1997].

TOSQL [Ariav 1986] extends SQL with the specification of a query's time aspects. The 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 relation's timestamp(s). S-reducibility is not supported. The built-in time-related processing is restricted to the above clauses and can be overwritten by stating the clauses explicitly, which makes TOSQL nonrestrictive and interval-preserving.

TSQL [Navathe and Ahmed 1987; 1989; 1993] is a superset of SQL, extending the latter with, for example, WHEN, TIME-SLICE, and MOVING WINDOW clauses. TSQL satisfies neither TUC (no adequate defaults for the above-mentioned clauses) nor S-reducibility. The restriction to

coalesced relation instances also breaks interval preservation. Apart from enforced coalescing, TSQL is nonrestrictive.

HSQL [Sarda 1990; 1993] is again a superset of SQL. The retrieval facilities are enhanced with facilities for coalescing (COALESCES, COALESCE ON), concurrent products, time-slicing (FROMTIME t1 TOTIME t2), and unfolding (EXPAND). The concurrent product provides built-in snapshot-reducible semantics for joins (and products), but not for subqueries, aggregates, set difference, and integrity constraints, for instance. 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 nonrestrictive. It does not support S-reducibility.

SQL-92 [Melton and Simon 1993] 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 temporal joins, for example). Because SQL-92 does not treat time with special semantics, it is trivially interval-preserving and nonrestrictive.

TempSQL [Bhargava and Gadia 1993; Gadia and Bhargava 1993; Gadia and Nair 1993] is an extension of SQL defined over relations where attribute values are temporal assignments, that is, partial functions from time into some value domain. A temporal expression ⟦. . . ⟧, which extracts the time domain of attribute values or relations, is a prominent feature of TempSQL. 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 [Gadia and Nair 1993]. 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 supported. TempSQL is restrictive in the sense that set operations have an enforced, built-in temporal semantics. TempSQL provides automatic coalescing, violating interval preservation.

IXSQL [Lorentzos and Mitsopoulos 1997] 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 nonrestrictive and, to some degree, it is also interval-preserving (intervals are not preserved by UNFOLD because interval boundaries are lost when unfolding a relation).

ChronoSQL [Böhlen and Marti 1994] was designed to show 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 and nonrestrictive.

TSQL2 [Snodgrass 1995] 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 nontemporal relation, SELECT SNAPSHOT has to be specified. TSQL2 neither satisfies TUC (valid-time relations are returned by default) nor S-reducibility (e.g., subqueries violate this [Böhlen et al. 1995]). TSQL2 is not interval-preserving because coalesced instances are enforced. While some operations come with built-in, implicit temporal semantics (e.g., set operations applied to temporal relations), TSQL2 is nonrestrictive because implicit timestamps can be rendered explicit.

SQL/TP [Toman 1998] utilizes instant timestamping in its semantics, thereby achieving sequenced semantics. Concerning TUC, the author states that "while SQL/TP itself does not literally follow these requirements, the compatibility can be easily achieved using a very simple syntactic manipulation of the source queries and adding tags to distinguish the particular compatibility modes" [Toman 1998, p. 214]. These "tags" could be the statement modifiers discussed in the present article. Extended S-reducibility and nonrestrictiveness are not satisfied because the intervals in the representation cannot be directly accessed by queries. Interval preservation is not guaranteed because timestamps are not specified.

The change proposals submitted to the SQL standardization committee [Snodgrass et al. 1996a; 1996b; 1998] 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 nondeterministic definition of the language. Nonrestrictiveness is achieved via nonsequenced statements. Extended S-reducibility is not satisfied because sequenced modifiers may only be applied to statements without explicit references to time dimensions.

7. SUMMARY AND RESEARCH DIRECTIONS

This article discusses how to extend an existing query language with temporal statement modifiers, enabling the language to better manage time-referenced, or temporal, information. We started by defining a number of syntactic and semantic desiderata, motivated by real-world concerns, that enable a temporal query language to contend with legacy applications, permit the coexistence of nontemporal and temporal data, and exploit the programmers' expertise in extending the existing nontemporal language. The desiderata are independent of any particular query language. No existing temporal language satisfies all of these desiderata. In particular, this article is the first to formulate requirements that combine salient features of temporal languages (snapshot-reducibility, temporal upward-

compatibility) with salient features of nontemporal languages (interval preservation, nonrestrictiveness).

As the next step, this article explores and exemplifies how these desiderata shape a temporal extension to SQL-92. 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 temporal query language that covers core as well as advanced language features, for example, views, integrity constraints, assertions, data definition, aggregation, and coalescing. The language supports both point- and interval-based semantics, with intervals preserved by default.

The introduction of statement modifiers makes it possible to obtain built-in time-related processing. The statement modifier's semantic approach is preferable to syntactic temporal extensions because statement modifiers ensure availability of built-in temporal support. The same (simple) statement modifier may be applied to an arbitrarily complex query to obtain built-in temporal processing. No other approaches known to the authors achieve temporal support using language extensions that are essentially independent of the complexity of the underlying nontemporal language being extended.

This article shows 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 the relational algebra. For this purpose, valid-time, transaction-time, and bitemporal counterparts of the standard relational algebra operators are provided. Finally, this article examines the properties of the extended language, verifying that the language satisfies the desiderata.

One interesting direction for future research is the application of this approach to other nontemporal query languages. ODMG [Cattell et al. 2000] would be a particularly appealing starting point, due to its prevalent use in practice.

It would also be useful to generalize statement modifiers to dimensions other than time—for example, spatial dimensions in spatial and spatio-temporal databases, the "dimensions" in data warehousing, or the new kinds of multidimensional data models. Providing general solutions that support the specific semantics associated with the new dimensions is an important challenge.

The notion of temporal upward compatibility assumes that the databases of existing DBMSs are extended with a temporal dimension only when the DBMS is replaced with a temporal DBMS. Other scenarios exist as well; indeed, many existing databases already record time-referenced data. For such databases to benefit from the built-in temporal support provided by the new DBMS, the time references must be recorded in the timestamp attributes with built-in semantics. How to (semiautomatically) migrate application code on a database that records its time references using regular attributes to a database where time references are captured in the special timestamp attributes remains an open problem. As we emphasized in Section 2.5, while temporal views offer a part of the solution, fully addressing this problem will require sophisticated tools.

Table IX. Definition of Constructs for Timestamp Manipulation

Syntax	*Semantics*
$[\![\texttt{PERIOD}\ `tp_1 - tp_2`]\!]_{temp}$	$\texttt{TIMESTAMP}`tp_1`$, $\texttt{TIMESTAMP}`tp_2`$
$[\![\texttt{FIRST}()]\!]_{temp}$	$\min(tp_1, tp_2)$
$[\![\texttt{LAST}(\texttt{TIMESTAMP}`tp_1`, \texttt{TIMESTAMP}`tp_2`)]\!]_{temp}$	$\max(tp_1, tp_2)$
$[\![\texttt{VTIME}(r)]\!]_{temp}$	$\texttt{TIMESTAMP}`r.VT^-`$, $\texttt{TIMESTAMP}`r.VT^+`$
$[\![\texttt{TTIME}(r)]\!]_{temp}$	$\texttt{TIMESTAMP}`r.TT^-`$, $\texttt{TIMESTAMP}`r.TT^+`$
$[\![\texttt{BEGIN}(per)]\!]_{temp}$	$[\![\texttt{FIRST}([\![per]\!]_{temp})]\!]_{temp}$
$[\![\texttt{END}(per)]\!]_{temp}$	$[\![\texttt{LAST}([\![per]\!]_{temp})]\!]_{temp}$
$[\![per_1\ \texttt{PRECEDES}\ per_2]\!]_{temp}$	$[\![\texttt{END}(per_1)]\!]_{temp} < [\![\texttt{BEGIN}(per_2)]\!]_{temp}$
$[\![per_1\ \texttt{MEETS}\ per_2]\!]_{temp}$	$[\![\texttt{END}(per_1)]\!]_{temp} = [\![\texttt{BEGIN}(per_2)^{-granule}1]\!]_{temp}$
$[\![per_1\ \texttt{OVERLAPS}\ per_2]\!]_{temp}$	$[\![\texttt{END}(per_1)]\!]_{temp} \geq [\![\texttt{BEGIN}(per_2)]\!]_{temp} \wedge [\![\texttt{END}(per_2)]\!]_{temp} \geq [\![\texttt{BEGIN}(per_1)]\!]_{temp}$
$[\![per_1\ \texttt{CONTAINS}\ per_2]\!]_{temp}$	$[\![\texttt{BEGIN}(per_2)]\!]_{temp} < [\![\texttt{BEGIN}(per_1)]\!]_{temp} \wedge [\![\texttt{END}(per_2)]\!]_{temp} \leq [\![\texttt{END}(per_1)]\!]_{temp}$
$[\![per + \texttt{INTERVAL}`iv`]\!]_{temp}$	$[\![\texttt{BEGIN}(per)]\!]_{temp} + iv$, $[\![\texttt{END}(per)]\!]_{temp}$
$[\![\texttt{INTERSECT}(per_1, per_2)]\!]_{temp}$	$\max([\![\texttt{BEGIN}(per_1)]\!]_{temp}, [\![\texttt{BEGIN}(per_2)]\!]_{temp})$, $\min([\![\texttt{END}(per_1)]\!]_{temp}, [\![\texttt{END}(per_2)]\!]_{temp})$
$[\![\texttt{DURATION}(per, granule)]\!]_{temp}$	$[\![\texttt{END}(per)]\!]_{temp}^{-granule} \geq [\![\texttt{BEGIN}(per)]\!]_{temp})$

Yet another future direction is the study of efficient implementation techniques. The current implementation of the temporal SQL proposed here illustrates the feasibility of a layered architecture [Torp et al. 1997]. This architecture can be used to identify bottlenecks in current database technology with respect to temporal database applications, and these findings may then prompt the development of new DBMS algorithms (e.g., as in Böhlen et al. [1996]).

Implementing functions, as in, for example, user-defined ADTs and PM/SQL modules, is another interesting research topic. Specifically, function calls are affected by statement modifiers, so that the semantics of a function call will depend on whether it is used in a temporal upward-compatible, a sequenced, or a nonsequenced context.

APPENDIX

A. PREDICATES AND FUNCTIONS ON TIMESTAMPS

Table IX gives a brief overview of the constructs used for timestamp manipulation. In the table, tp and iv, possibly indexed, denote a time point of type `TIMESTAMP` [Melton and Simon 1993] and a time duration of type `INTERVAL` [Melton and Simon 1993], respectively. Also, p is shorthand for `PERIOD` $'tp_1 - tp_2'$ and $granule \in$ {`YEAR`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, `SECOND`} denotes a granularity.

Table X. Time-Slice Operators on Nontemporal Relations

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

B. AUXILIARY ALGEBRAIC OPERATORS

Tables X and XI define the auxiliary operators that time-slice relations and turn timestamps into regular, explicit attributes. Unlike the other algebraic operators defined in this article, these operators are overloaded to apply to valid-time, transaction-time, and bitemporal relations, meaning that the type of an argument determines the operation performed. This property was exploited to more concisely define the semantics of core statements in Table III.

The functions have variants for both valid and transaction times. For example, the valid-time version of the first time-slice operation, τ^{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 time-slice operation, δ_{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. BITEMPORAL RELATIONAL ALGEBRA

As the valid-time algebra is 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 differs from this algebra only in that it deals with bitemporal rectangles rather than periods. Bitemporal selection, projection, set union, and Cartesian product (see Figure 4) are straightforward extensions.

Bitemporal difference is substantially more complex. It is defined in terms of three auxiliary predicates [Böhlen and Jensen 1996]. The idea behind the operator's definition is illustrated in Figure 5, 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.

The determining time lines associated with r_2-tuples play a crucial role in splitting r_1-tuples, and thus in defining the result tuples. The determining time lines start at each vertex (corner) of an r_2-tuple, and extend until

Table XI. Time-Slice and Snapshot Operators on Temporal Relations

Function	Semantics if r is a valid-time relation
$\tau_{tp}^{vt}(r)$	$\{\langle t\rangle \mid \exists VT(\langle t\Vert VT\rangle \in r \wedge VT\ overlaps\ tp)\}$
$\tau_{tp}^{tt}(r)$	r
$\delta_{per}^{vt}(r)$	$\{\langle t\Vert VT\rangle \mid \exists VT'(\langle t\Vert VT'\rangle \in r \wedge VT'\ overlaps\ per \wedge VT = intersect(VT', per))\}$
$\delta_{per}^{tt}(r)$	r
$SN^{vt}(r)$	$\{\langle t, VT\rangle \mid \langle t\Vert VT\rangle \in r\}$
$SN^{tt}(r)$	r
Function	**Semantics if r is a transaction-time relation**
$\tau_{tp}^{vt}(r)$	r
$\tau_{tp}^{tt}(r)$	$\{\langle t\rangle \mid \exists TT(\langle t\Vert TT\rangle \in r \wedge VT\ overlaps\ tp)\}$
$\delta_{per}^{vt}(r)$	r
$\delta_{per}^{tt}(r)$	$\{\langle t\Vert TT\rangle \mid \exists TT'(\langle t\Vert TT'\rangle \in r \wedge TT'\ overlaps\ per \wedge TT = intersect(TT', per))\}$
$SN^{vt}(r)$	r
$SN^{tt}(r)$	$\{\langle t, TT\rangle \mid \langle t\Vert TT\rangle \in r\}$
Function	**Semantics if r is a bitemporal relation**
$\tau_{tp}^{vt}(r)$	$\{\langle t\Vert TT\rangle \mid \exists VT(\langle t\Vert VT, TT\rangle \in r \wedge VT\ overlaps\ tp)\}$
$\tau_{tp}^{tt}(r)$	$\{\langle t\Vert VT\rangle \mid \exists TT(\langle t\Vert VT, TT\rangle \in r \wedge TT\ overlaps\ tp)\}$
$\delta_{per}^{vt}(r)$	$\{\langle t\Vert VT, TT\rangle \mid \exists VT'(\langle t\Vert VT', TT\rangle \in r \wedge VT'\ overlaps\ per \wedge VT = intersect(VT', per))\}$
$\delta_{per}^{tt}(r)$	$\{\langle t\Vert VT, TT\rangle \mid \exists TT'(\langle t\Vert VT, TT'\rangle \in r \wedge TT\ overlaps\ per \wedge TT = intersect(TT', per))\}$
$SN^{vt}(r)$	$\{\langle t, VT\Vert TT\rangle \mid \langle t\Vert VT, TT\rangle \in r\}$
$SN^{tt}(r)$	$\{\langle t, TT\Vert VT\rangle \mid \langle t\Vert VT, TT\rangle \in r\}$

$$
\begin{aligned}
\sigma_c^{bi}(r) &\triangleq \{\langle t\Vert VT,TT\rangle \mid \langle t\Vert VT,TT\rangle \in r \wedge c(\langle t\Vert VT,TT\rangle)\}\\
\pi_f^{bi}(r) &\triangleq \{\langle t_1\Vert VT,TT\rangle \mid \exists t_2\ (\langle t_2\Vert VT,TT\rangle \in r \wedge t_1 = f(\langle t_2\Vert VT,TT\rangle))\}\\
r_1 \cup^{bi} r_2 &\triangleq \{\langle t\Vert VT,TT\rangle \mid \langle t\Vert VT,TT\rangle \in r_1 \vee \langle t\Vert VT,TT\rangle \in r_2\}\\
r_1 \times^{bi} r_2 &\triangleq \{\langle\langle t_1, VT_1, TT_1\rangle \circ \langle t_2, VT_2, TT_2\rangle \Vert VT,TT\rangle \mid\\
&\qquad \langle t_1\Vert VT_1,TT_1\rangle \in r_1 \wedge \langle t_2\Vert VT_2,TT_2\rangle \in r_2 \wedge\\
&\qquad VT = intersect(VT_1, VT_2)\ \wedge TT = intersect(TT_1, TT_2) \wedge\\
&\qquad VT_1\ overlaps\ VT_2 \wedge TT_1\ overlaps\ TT_2\}\\
r_1 \setminus^{bi} r_2 &\triangleq \{\langle t\Vert VT,TT\rangle \mid \exists VT_1, TT_1(\langle t\Vert VT_1,TT_1\rangle \in r_1 \wedge\\
&\qquad candidate_tuple(t, VT_1, TT_1, r_2, VT, TT) \wedge\\
&\qquad non_overlapping(t, VT, TT, r_2) \wedge\\
&\qquad unsplittable(t, VT_1, TT_1, VT, TT, r_2))
\end{aligned}
$$

Fig. 4. Bitemporal algebra.

they are blocked by a value-equivalent r_2-tuple or reach the border of the r_1-tuple. The definition in Figure 4 identifies three requirements for a result tuple X.

(1) The time coordinates of X are derived either from the time coordinates of an r_1-tuple or from an overlapping r_2-tuple with a determining line

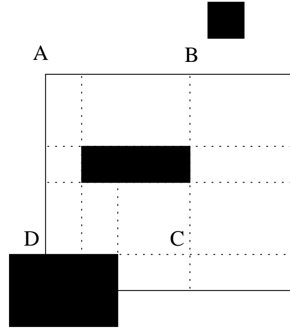

Fig. 5. Bitemporal difference.

that defines a time coordinate of X. The predicate *candidate_tuple*$(t, VT_1, TT_1, r_2, VT, TT)$ is used to ensure that VT and TT satisfy these restrictions with respect to t, VT_1, TT_1, and r_2.

(2) X does not temporally overlap with any value-equivalent r_2-tuple. The predicate *non_overlapping*(t, VT, TT, r_2) ensures this.

(3) No determining time lines defined by r_2-tuples that are value-equivalent to X cross its time rectangle, that is, *unsplittable*$(t, VT_1, TT_1, VT, TT, r_2)$.

The formal definitions of these predicates can be found elsewhere [Böhlen and Jensen 1996].

The final operation is the coalescing of bitemporal relations. Transaction-time coalescing, $coal_{tt}^{bi}$, guarantees maximal transaction-time periods, and is illustrated in Figure 6. The five white rectangles illustrate the times of five value-equivalent tuples in the uncoalesced relation. The gray tuple is one of the tuples resulting from coalescing in transaction time. Transaction-time coalescing ensures (a) maximal expansion in the transaction-time dimension and (b) no coalescing in the valid-time dimension. 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 time and transaction time are reversed. The formal definitions of coalescing bitemporal relations are given in Böhlen and Jensen [1996].

D. PROOF OF THEOREM 5.1

To prove Theorem 5.1 (S-reducibility of sequenced queries), we consider each equivalence in turn. We use the definitions in Tables VII, VIII, and XI to substitute algebraic expressions to prove their equivalence.

Selection:

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

$$\sigma_c(\tau_{tp}^{vt}(r)) = \{t \mid \langle t \| VT\rangle \in r \wedge VT \textit{ overlaps } tp \wedge c(t)\}$$

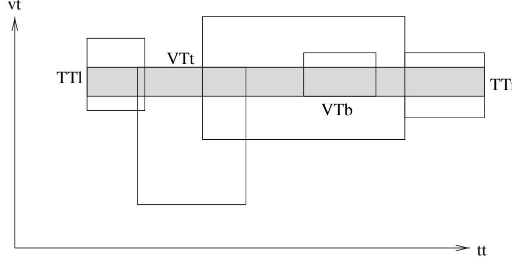

Fig. 6. Transaction time for coalescing a bitemporal relation.

To show that these definitions are equivalent, we first exploit the commutativity of conjunction to rewrite " $VT\ overlaps\ tp \wedge c(t)$" to " $c(t) \wedge VT\ overlaps\ tp$." Since the formulation of the theorem disallows the use of VT in predicate c, $c(\langle t, VT\rangle)$ and $c(t)$ are equivalent, the equivalence follows.

Projection:

$$\tau^{vt}_{tp}(\pi^{vt}_{f}(r)) = \{t_1 \mid \exists t_2(\langle t_2 \| VT\rangle \in r \wedge t_1 = f(\langle t_2, VT\rangle)) \wedge VT\ overlaps\ tp\}$$

$$\pi_f(\tau^{vt}_{tp}(r)) = \{t_1 \mid \exists t_2(\langle t_2 \| VT\rangle \in r \wedge VT\ overlaps\ tp \wedge t_1 = f(t_2))\}$$

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

Union:

$$\tau^{vt}_{tp}(r_1 \cup^{vt} r_2) = \{t \mid (\langle t \| VT\rangle \in r_1 \vee \langle t \| VT\rangle \in r_2) \wedge VT\ overlaps\ tp\}$$

$$\tau^{vt}_{tp}(r_1) \cup \tau^{vt}_{tp}(r_2) = \{t \mid (\langle t \| VT\rangle \in r_1 \wedge VT\ overlaps\ tp) \vee (\langle t \| VT\rangle \in r_2 \wedge VT\ overlaps\ tp)\}$$

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

Cartesian product:

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

$$\tau^{vt}_{tp}(r_1) \times_c \tau^{vt}_{tp}(r_2) = \{t_1 \circ t_2 \mid \langle t_1 \| VT_1\rangle \in r_1 \wedge VT_1\ overlaps\ tp \wedge \langle t_2 \| VT_2\rangle \in r_2 \wedge VT_2\ overlaps\ tp\}$$

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.

$$\begin{aligned}
&VT_1 \textit{ overlaps } VT_2 \wedge VT = \textit{intersect}\ (VT_1, VT_2) \wedge VT \textit{ overlaps } tp \\
&\qquad \Downarrow \text{(elimination of } VT) \\
&VT_1 \textit{ overlaps } VT_2 \wedge \textit{intersect}\ (VT_1, VT_2) \textit{ overlaps } tp \\
&\qquad \Downarrow \text{(replace periods with points, cf. Table IX)} \\
&VT_1^+ \geq VT_2^- \wedge VT_2^+ \geq VT_1^- \wedge \max(VT_1^-, VT_2^-) < tp \wedge \min(VT_1^+, VT_2^+) > tp \\
&\qquad \Downarrow (\max(A, B) \leq C \equiv A \leq C \wedge B \leq C), (\min(A, B) \geq C \equiv A \geq C \wedge B \geq C) \\
&VT_1^+ \geq VT_2^- \wedge VT_2^+ \geq VT_1^- \wedge VT_1^- \leq tp \wedge VT_2^- \leq tp \wedge VT_1^+ \geq tp \wedge VT_2^+ \geq tp
\end{aligned}$$

Next, we rewrite the second formula.

$$\begin{aligned}
&VT_1 \textit{ overlaps } tp \wedge VT_2 \textit{ overlaps } tp \\
&\qquad\qquad \Downarrow \text{(replace periods with points, cf. Table IX)} \\
&VT_1^- \leq tp \wedge VT_1^+ \geq tp \wedge VT_2^- \leq tp \wedge VT_2^+ \geq tp \\
&\qquad\qquad \Downarrow (A \leq C \wedge B \geq C \Rightarrow B \geq A) \\
&VT_1^- \leq tp \wedge VT_1^+ \geq tp \wedge VT_2^- \leq tp \wedge VT_2^+ \geq tp \wedge \ VT_1^+ \geq VT_2^- \wedge VT_2^+ \geq VT_1^-
\end{aligned}$$

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

Difference:

$$\tau_{tp}^{vt}(r_1 \setminus^{vt} r_2) = \{t \mid \phi_1\}$$

$$\tau_{tp}^{vt}(r_1) \setminus \tau_{tp}^{vt}(r_2) = \{t \mid \phi_2\}$$

where ϕ_1 is defined as

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

and ϕ_2 is defined as

$$\exists VT_1(\langle t \| VT_1 \rangle \in r_1 \wedge VT_1 \textit{ overlaps } tp) \wedge \neg \exists VT_2(\langle t \| VT_2 \rangle \in r_2 \wedge VT_2 \textit{ overlaps } tp).$$

To prove the two sets equivalent, we have to show that the defining formulas are equivalent, that is, $\phi_1 \equiv \phi_2$. We do so by proving two implications $\phi_1 \Rightarrow \phi_2$ and $\phi_1 \Leftarrow \phi_2$ in turn. The step-by-step proof can be found in Böhlen and Jensen [1996].

ACKNOWLEDGMENTS

We greatly appreciate the contributions of Renato Busatto, Robert Marti, and Andreas Steiner. Renato Busatto contributed to the formalization of interval preservation, bitemporal negation, and the proof of Theorem 5.1. Robert Marti took part in early work on statement modifiers. Andreas Steiner implemented a core part of this language. We also appreciate Won Kim's extensive comments, which significantly improved the paper.

REFERENCES

ALLEN, J. F. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM 26*, 11 (Nov.), 832–843.

ARIAV, G. 1986. A temporally oriented data model. *ACM Trans. Database Syst. 11*, 4 (Dec.), 499–527.

BAIR, J., JENSEN, C. S., SNODGRASS, R. T., AND BOEHLEN, M. 1997. Notions of upward compatibility of temporal query languages. *Business Inf. 39*, 1 (Feb.), 25–34.

BHARGAVA, G. AND GADIA, S. K. 1993. Relational database systems with zero information loss. *IEEE Trans. Knowl. Data Eng. 5*, 1 (Feb.), 76–87.

BÖHLEN, M. H., BUSATTO, R., AND JENSEN, C. S. 1998a. Point- versus interval-based temporal data models. In *Proceedings of the 14th IEEE International Conference on Data Engineering* (Orlando, FL, Feb. 23-27). IEEE Computer Society Press, Los Alamitos, CA, 192–200.

BÖHLEN, M. H., JENSEN, C. S., AND SKJELLAUG, B. 1998b. Spatio-temporal database support for legacy applications. In *Proceedings of the 1998 ACM Symposium on Applied Computing* (Atlanta, GA, Feb.). ACM Press, New York, NY, 226–234.

BÖHLEN, M. H. AND JENSEN, C. S. 1996. Seamless integration of time into SQL. Tech. Rep. R-96-2049. Aalborg Univ., Aalborg, Denmark.

BÖHLEN, M. H., SNODGRASS, R. T., AND SOO, M. D. 1996. Coalescing in temporal databases. In *Proceedings of the 22nd International Conference on Very Large Data Bases* (VLDB '96, Bombay, Sept.). Morgan Kaufmann Publishers Inc., San Francisco, CA, 180–191.

BÖHLEN, M. H., JENSEN, C. S., AND SNODGRASS, R. T. 1995. Evaluating the completeness of TSQL2. In *Proceedings of the International Workshop on Recent Advances in Temporal Databases* (Zurich, Sept.), S. Clifford and A. Tuzhlin, Eds. Springer-Verlag, New York, NY, 153–172.

BÖHLEN, M. AND MARTI, R. 1994. On the completeness of temporal database query languages. In *Proceedings of the First International Conference on Temporal Logic* (ICTL'94, Bonn, July). Springer-Verlag, Secaucus, NJ, 283–300.

BÖHLEN, M. 1994. Managing temporal knowledge in deductive databases. Ph.D. Dissertation. ETH, Zurich, Switzerland.

CATTELL, R. G. G., BARRY, D. K., BERLER, M., EASTMAN, J., JORDAN, D., RUSSELL, C., SCHADOW, O., STANIENDA, T., AND VELEZ, F. 2000. *The Object Data Standard OMDG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

CLIFFORD, J., CROKER, A., AND TUZHILIN, A. 1993. *On the Completeness of Query Languages for Grouped and Ungrouped Historical Data Models*. Benjamin/Cummings, Redwood City, CA.

CLIFFORD, J., DYRESON, C., ISAKOWITZ, T., JENSEN, C. S., AND SNODGRASS, R. T. 1997. On the semantics of NOW in databases. *ACM Trans. Database Syst. 22*, 2, 171–214.

CELKO, J. 1995. *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

CERI, S. AND GOTTLOB, G. 1985. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr.), 324–345.

GADIA, S. K. AND BHARGAVA, G. 1993. SQL-like seamless query of temporal data. In *Proceedings of the International Workshop on Infrastructure for Temporal Databases* (Arlington, TX, June), R. T. Snodgrass, Ed.

GADIA, S. K. AND NAIR, S. S. 1993. Temporal databases: A prelude to parametric data. In *Temporal Databases: Theory, Design, and Implementation*, A. Tansel, J. Clifford, S. Gadia,

S. Jajodia, A. Segev, and R. Snodgrass, Eds. Benjamin/Cummings, Redwood City, CA, 28–66.

GADIA, S. K. 1988. A homogeneous relational model and query languages for temporal databases. *ACM Trans. Database Syst. 13*, 4 (Dec.), 418–448.

GADIA, S. K. 1986. Weak temporal relations. In *Proceedings of the ACM Symposium on Principles of Database Systems* (PODS '86, Cambridge, MA). ACM Press, New York, NY.

DYRESON, C., GRANDI, F., KÄFER, W., KLINE, N., LORENTZOS, N., MITSOPOULOS, Y., MONTANARI, A., NONEN, D., PERESSI, E., PERNICI, B., RODDICK, J. F., SARDA, N. L., SCALAS, M. R., SEGEV, A., SNODGRASS, R. T., SOO, M. D., TANSEL, A., TIBERIO, P., AND WIEDERHOLD, G. 1994. A consensus glossary of temporal database concepts. *SIGMOD Rec. 23*, 1 (Mar.), 52–64.

JENSEN, C. S. AND SNODGRASS, R. T. 1999. Temporal data management. *IEEE Trans. Knowl. Data Eng. 11*, 1 (Jan./Feb.), 36–44.

JENSEN, C. S., SOO, M. D., AND SNODGRASS, R. T. 1994. Unifying temporal data models via a conceptual model. *Inf. Syst. 19*, 7 (Oct.), 513–547.

LORENTZOS, N. A. AND MITSOPOULOS, Y. G. 1997. SQL extension for interval data. *IEEE Trans. Knowl. Data Eng. 9*, 3 (May/June), 480–499.

MCKENZIE, L. E. AND SNODGRASS, R. T. 1991. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Comput. Surv. 23*, 4 (Dec.), 501–543.

MELTON, J. 1999. Information technology, database languages, SQL.

MELTON, J. 1992. Information technology, database languages, SQL.

MELTON, J. AND SIMON, A. R. 1993. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers Inc., San Francisco, CA.

NAIR, S. AND GADIA, S. 1993. Algebraic optimization in a relational model for temporal databases. In *Proceedings of the International Workshop on Infrastructure for Temporal Databases* (Arlington, TX, June), R. T. Snodgrass, Ed.

NAVATHE, S. B. AND AHMED, R. 1993. Temporal extensions to the relational model and SQL. In *Temporal Databases: Theory, Design, and Implementation*, A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, Eds. Benjamin/Cummings, Redwood City, CA, 92–109.

NAVATHE, S. B. AND AHMED, R. 1989. A temporal relational model and a query language. *Inf. Sci. 49*, 1, 2 & 3 (Oct./Nov./Dec.), 147–175.

NAVATHE, S. B. AND AHMED, R. 1987. TSQL-A language interface for history databases. In *Proceedings of the Conference on Temporal Aspects in Information Systems* (May). Assn. Francaise pour Cybern. Econ. Tech., Montreuil, France, 113–128.

SARDA, N. 1993. HSQL: A historical query language. In *Temporal Databases: Theory, Design, and Implementation*, A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, Eds. Benjamin/Cummings, Redwood City, CA, 110–140.

SARDA, N. L. 1990. Algebra and query language for a historical data model. *Computer J. 33*, 1 (Feb.), 11–18.

SCHUELER, B. 1977. Update reconsidered. In *Architecture and Models in Data Base Management Systems*, G. M. Nijssen, Ed. North-Holland Publishing Co., Amsterdam, The Netherlands.

SNODGRASS, R. T. 2000. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

SNODGRASS, R. T., BÖHLEN, M. H., JENSEN, C. S., AND STEINER, A. 1998. Transitioning temporal support in TSQL2 to SQL3. In *Temporal Databases: Research and Practice: State-of-the-Art Survey*, O. Etzion, S. Jajodia, and S. Sripada, Eds. Springer-Verlag, New York, NY, 150–194.

SNODGRASS, R. T., BÖHLEN, M. H., JENSEN, C. S., AND KLINE, N. 1996a. Adding valid time to SQL/Temporal. ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2.

SNODGRASS, R. T., BÖHLEN, H., JENSEN, S., AND STEINER, A. 1996b. Adding transaction time to SQL/Temporal: Temporal change proposal. ANSI X3H2-96-152r, ISO-;ANSI SQL/ISO/IECJTC1/SC21/WG3 DBL MCI-143.

SNDOGRASS, R. T., ED. 1995. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Hingham, MA.

SNODGRASS, R. T. 1993. An overview of TQuel. In *Temporal Databases: Theory, Design, and Implementation*, A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, Eds. Benjamin/Cummings, Redwood City, CA, 141–182.

SNODGRASS, R. T. 1990. Temporal databases status and research directions. *SIGMOD Rec. 19*, 4 (Dec.), 83–89.

SNODGRASS, R. T. 1987. The temporal query language TQuel. *ACM Trans. Database Syst. 12*, 2 (June), 247–298.

SNODGRASS, R. T. AND AHN, I. 1985. A taxonomy of time in databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. ACM Press, New York, NY, 236–246.

SOO, M. D., JENSEN, C. J., AND SNODGRASS, T. 1995. An algebra for TSQL2. In *The TSQL2 Temporal Query Language*, R. T. Sndograss, Ed. Kluwer Academic Publishers, Hingham, MA, 505–546.

TANSEL, A., CLIFFORD, J., GADIA, S., JAJODIA, S., SEGEV, A., AND SNODGRASS, R., EDS. 1993. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, Redwood City, CA.

TOMAN, D. AND NIWINSKI, D. 1996. First-order queries over temporal databases inexpressible in temporal logic. In *Proceedings of the Fifth International Conference on Extending Database Technology* (Avignon, France). Springer-Verlag, New York, NY, 307–324.

TOMAN, D. 1998. Point-based temporal extensions of SQL and their efficient implementation. In *Temporal Databases: Research and Practice: State-of-the-Art Survey*, O. Etzion, S. Jajodia, and S. Sripada, Eds. Springer-Verlag, New York, NY, 211–237.

TORP, K., JENSEN, C. S., AND BÖHLEN, M. H. 1997. Layered implementation of Temporal DBMSs: Concepts and techniques. In *Proceedings of the 5th International Conference on Database Systems for Advanced Applications* (Melbourne, Australia, Apr.), R. Topor and K. Tanaka, Eds. World Scientific Publishing Co., Inc., River Edge, NJ.

TSICHRITZIS, D. C. AND LOCHOVSKY, F. H. 1982. *Data Models*. Prentice-Hall, New York, NY.

VAN BENTHEM, J. 1991. *The Logic of Time*. 2nd ed. Kluwer Academic Publishers, Hingham, MA.

VAN GELDER, A. AND TOPOR, R. W. 1991. Safety and translation of relational calculus queries. *ACM Trans. Database Syst. 16*, 2 (June), 235–278.

WIEDERHOLD, G. 1973. How to write a schema for a time oriented medical record data bank. Tech. Rep. Stanford University, Stanford, CA.

ZANIOLO, C., CERI, S., FALOUTSOS, C., SNODGRASS, R. T., SUBRAHMANIAN, V. S., AND ZICARI, R. 1997. *Advanced Database Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Received: April 1998; revised: April 2000; accepted: September 2000