Transitioning Temporal Support in TSQL2 to SQL3

Richard T. Snodgrass[1], Michael H. Böhlen[2], Christian S. Jensen[2], and Andreas Steiner[3]

[1] Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, rts@cs.arizona.edu
[2] Department of Mathematics and Computer Science, Aalborg University, Fredrik Bajers Vej 7E, DK–9220 Aalborg Ø, DENMARK, {boehlen,csj}@cs.auc.dk
[3] Institut fur Informationssysteme, ETH Zentrum, CH-8092 Zurich, Switzerland, steiner@inf.ethz.ch

Abstract. This document summarizes the proposals before the SQL3 committees to allow the addition of tables with valid-time and transaction-time support into SQL/Temporal, and explains how to use these facilities to migrate smoothly from a conventional relational system to one encompassing temporal support. Initially, important requirements to a temporal system that may facilitate such a transition are motivated and discussed. The proposal then describes the language additions necessary to add valid-time support to SQL3 while fulfilling these requirements. The constructs of the language are divided into four levels, with each level adding increased temporal functionality to its predecessor. A prototype system implementing these constructs on top of a conventional DBMS is publicly available.

1 Introduction

We introduce constructs that have been submitted to the ISO SQL3 committee as change proposals to SQL/Temporal [8] to add valid-time and transaction-time support to SQL3 [14, 15]. These constructs are variants of those first defined in TSQL2 [13].

While temporal database research has a long history (cf. [17]), momentum for a language designed with input from a substantial part of the community first arose at a 1993 temporal infrastructure workshop [9]. The TSQL2 committee was subsequently formed in July, 1993 in response to a general invitation sent to the community. This committee consisted of Richard T. Snodgrass, Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Christian S. Jensen, Ramez Elmasri, Fabio Grandi, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, Ting Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Surynarayana M. Sripada. The committee produced a preliminary

O. Etzion, S. Jajodia, and S. Sripada (Eds.): Temporal Databases - Research and Practice
LNCS 1399, pp. 150–194, 1998.

language specification the following January, which appeared in the *ACM SIGMOD Record* [10]. Based on responses to that specification, changes were made to the language, and the final language specification and 28 commentaries were made available via anonymous FTP in early October, 1994. A book describing the language and examining in detail the design decisions [13] was released at the VLDB International Workshop on Temporal Databases in September, 1995.

Richard Snodgrass started working with the ANSI and ISO SQL3 committees in late 1994. The first step was to propose a new part to SQL3, termed SQL/Temporal [12]. This was formally approved in July, 1995. Jim Melton agreed to edit this new part.

Discussions then commenced on adding valid-time support to SQL/Temporal. While the ANSI committee was supportive of the overall approach, there were several concerns voiced about the TSQL2 design. The major objections were as follows.

1. Rows in TSQL2 are timestamped with temporal elements [4, 7], which are sets of periods, each of which extends from a starting instant to an ending instant. Temporal elements are not bounded in size, which means that all timestamped rows will also be unbounded in size.
2. Duplicates are not supported: TSQL2 disallows value-equivalent rows, and temporal element timestamps, being sets, also do not permit duplicates. The analogy is with the relational algebra, which is also based on sets, and hence does not accommodate duplicates.
3. A table with temporal support is returned with a conventional `SELECT` statement. To get a table without temporal support, the `SNAPSHOT` keyword is required. The committee felt that a conventional query should return a table without temporal support.
4. There was no formal semantics for TSQL2.
5. There existed no implementation of the proposed constructs.
6. The keywords `VALID` and `TRANSACTION` were judged to be too generic.

After many discussions with the committee and with others, the following solutions were agreed upon. This process took well over a year to complete. These modifications are reasonable, as the TSQL2 design and the change proposals had differing objectives.

1. Rows would be timestamped with periods rather than temporal elements. This enabled timestamps to be bounded in size.
2. Value-equivalent rows would be permitted, so that duplicates could be accommodated.
3. `SNAPSHOT` was discarded. A conventional query returns a table with no temporal support (this was later generalized to the highly desirable property of temporal upward compatibility [1]). The `VALID` clause was moved to before the `SELECT` and later generalized to support sequenced queries (which were developed as part of the ATSQL design [3]).
4. A formal semantics for the language was developed [3].

5. Michael Böhlen and Andreas Steiner produced a public domain prototype implementation. Andreas has continued to evolve this prototype to be consistent with the change proposals.
6. The keywords were changed to `VALIDTIME` and `TRANSACTIONTIME`.

Many other smaller changes were made to the language proposals and to the wording of the change proposals to address concerns of the committee members. The full story, including the change proposals themselves, can be found at `FTP.cs.arizona.edu/tsql/tsql2/sql3`.

The change proposals have been unanimously approved by the ANSI SQL3 committee (ANSI X3H2) and are under consideration by the ISO SQL3 committee (ISO/IEC JTC 1/SC 21/WG 3 DBL).

In this paper, we first outline a four-level approach for the integration of time. The language extensions are fairly minimal. Each level is described via a quick tour consisting of a set of examples. These examples have been tested in a prototype which is publicly available [16]. We examine valid-time support first, then consider transaction-time and bitemporal support.

2 The Problem

Most databases store time-varying information. For such databases, SQL is often the language of choice for developing applications that utilize the information in these databases. However, users also realize that SQL does not provide adequate support for temporal applications. To illustrate this, the reader is invited to attempt to formulate the following straightforward, realistic statements in SQL3. An intermediate SQL programmer can express all of them in SQL for a non-time-varying database in perhaps five minutes. However, even SQL experts find these same queries challenging to do in several *hours* when time-varying data is taken into account.

- An `employee` table has five columns: `name`, `eno`, `street`, `city`, and `birthdate`. The related `salary` table has two columns: `eno` and `amount` (as a monthly salary). We then store historical information in both tables by adding a column, `When`, of data type `PERIOD`. Column `salary.eno` is a foreign key for `employee.eno`. This means that at each point in time, the integer value in the `salary.eno` column also occurs in the `eno` column of `employee` at the same time. This cannot be expressed via SQL's foreign key constraint, which does not take time into account. The reader is invited to attempt to formulate this constraint instead as an assertion.
- Consider the query "List those employees who have no salary." This can easily be expressed in SQL, using `EXCEPT` or `NOT EXISTS`, on the original table. Things are just a little harder with the `When` column; a `WHERE` predicate is required to extract the current employees and current salaries. Now formulate the query "List those employees who have no salary, and indicate when." `EXCEPT` and `NOT EXISTS` will not work, because they do not consider time. This simple temporal query is challenging even to SQL experts.

- Consider the query "Give the number of employees making over $5,000 in each city." Again, this is a simple query in SQL on the original table. We invite the reader to formulate the query "Give *the history of* the number of employees making over $5,000 in each city" on the table with the `When` column. This query is extremely difficult without temporal support in the language. One approach is to expand each row in both tables into all the days that it was valid, then count up the employees for each day. However, we would like a solution that did not force such an expansion, and also used the periods directly, as that approach is likely to be more efficient than a "point-based" expansion would be.
- Now formulate the modification "Give Therese a salary of $6,000 for 1994." This modification is difficult in SQL because only a portion of many validity periods needs be changed, with the information outside of 1994 retained.

Most users know only too well that while SQL is an extremely powerful language for writing queries on the current state, the language provides much less help when writing temporal queries, modifications, and constraints.

3 Outline of the Solution

The problem with formulating these SQL statements is due in large part to the extreme difficulty of specifying in SQL the correct values of the timestamp column(s) of the result. The solution is to allow the DBMS to compute these values, moving the complexity from the application code into the DBMS. With the language extensions proposed here, the above queries can all be easily written by an intermediate SQL programmer in a few minutes. We provide these SQL statements here; the language constructs will be explained and exemplified in detail in the remainder of the paper.

Both tables with valid-time support and temporal referential integrity can be specified using the `VALIDTIME` reserved word.

```
CREATE TABLE employee(ename VARCHAR(12),
          eno} INTEGER VALIDTIME PRIMARY KEY,
          street VARCHAR(22), city VARCHAR(10), birthday DATE)
AS VALIDTIME PERIOD(DATE)

CREATE TABLE salary(eno INTEGER VALIDTIME PRIMARY KEY
                    VALIDTIME REFERENCES employee,
          amount INTEGER)
AS VALIDTIME PERIOD(DATE)
```

Here we indicate that the table has valid-time support through "`AS VALIDTIME PERIOD(DATE)`" and that the integrity constraints (primary key, referential integrity) are to hold for each point in time (day) through "`VALIDTIME PRIMARY KEY`" and "`VALIDTIME REFERENCES`."

For the query "List those employees who have no salary," we are interested only in the current employees. We use temporal upward compatibility to extract this information from the historical information stored in the employee table.

```
SELECT ename
FROM employee
WHERE eno NOT IN (SELECT eno FROM salary)
```

This results in a conventional table, with one column.

We use sequenced valid semantics in the query "List those employees who had no salary, and when."

```
VALIDTIME SELECT ename
FROM employee
WHERE eno NOT IN (SELECT eno FROM salary)
```

The added "VALIDTIME" reserved word specifies that the query is to be evaluated at each point in time. At some times, an employee may not have a salary, whereas at other times, the employee may have a salary. A one-column table results, but now with valid-time support (i.e., the periods of time when each employee did not have a salary are included).

The query "Give the number of highly paid employees in each city" is easy, given temporal upward compatibility.

```
SELECT city, COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
GROUP BY city
```

Again, we just get the current count for each city, i.e., the number of employees now in each city. To extract "*the history of* the number of highly-paid employees in each city," only a simple change is required.

```
VALIDTIME SELECT city, COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
GROUP BY city
```

For each city, a time-varying count will be returned.

Modifications work in similar ways. The modification "Give Therese a salary of $6,000 for 1994" can be expressed by following VALIDTIME with a period expression.

```
VALIDTIME PERIOD '[1994-01-01 - 1994-12-31]' UPDATE salary
SET amount = 6000
WHERE eno IN (SELECT eno FROM employee WHERE ename = 'Therese')
```

Here again, we exploit our knowledge of SQL to first write the update ignoring time, then change it in minor ways to take account of time.

These statements are reminiscent of the kinds of SQL statements that application programmers are called to write all the time. The potential for increased productivity is dramatic. Statements that previously took hours to write, or were simply too difficult to express, can take only minutes to write with the extensions discussed here.

We now return to the important question of migrating legacy databases. In the next section, we formulate several requirements of SQL/Temporal to allow graceful migration of applications from conventional to temporal databases.

4 Migration

The potential users of temporal database technology are enterprises with applications[1] that need to manage potentially large amounts of time-varying information. These include financial applications such as portfolio management, accounting, and banking; record-keeping applications, including personnel, medical records, and inventory; and travel applications such as airline, train, and hotel reservations and schedule management. It is most realistic to assume that these enterprises are already managing time-varying data and that the temporal applications are already in place and working. Indeed, the uninterrupted functioning of applications is likely to be of vital importance.

For example, companies usually have applications that manage the personnel records of their employees. These applications manage large quantities of time-varying data, and they may benefit substantially from built-in temporal support in the DBMS [11]. Temporal queries that are shorter and more easily formulated are among the potential benefits. This leads to improved productivity, correctness, and maintainability.

This section explores the problems that may occur when migrating database applications from an existing to a new DBMS, and it formulates a number of requirements [1] to the new DBMS that must be satisfied in order to avoid different potential problems when migrating. We proceed by identifying four successively more general levels of queries and modifications.

4.1 Upward Compatibility

Perhaps the most important aspect of ensuring a smooth transition is to guarantee that all application code without modification will work with the new system exactly with the same functionality as with the existing system.

To explore the relationship between nontemporal and temporal data and queries, we employ a series of figures that demonstrate increasing query and update functionality. In Fig. 1, a conventional table is denoted with a rectangle.

[1] We use "database application" non-restrictively, for denoting any software system that uses a DBMS as a standard component.

The current state of this table is the rectangle in the upper-right corner. Whenever a modification is made to this table, the previous state is discarded; hence, at any time only the current state is available. The discarded prior states are denoted with dashed rectangles; the right-pointing arrows denote the modification that took the table from one state to the next state.

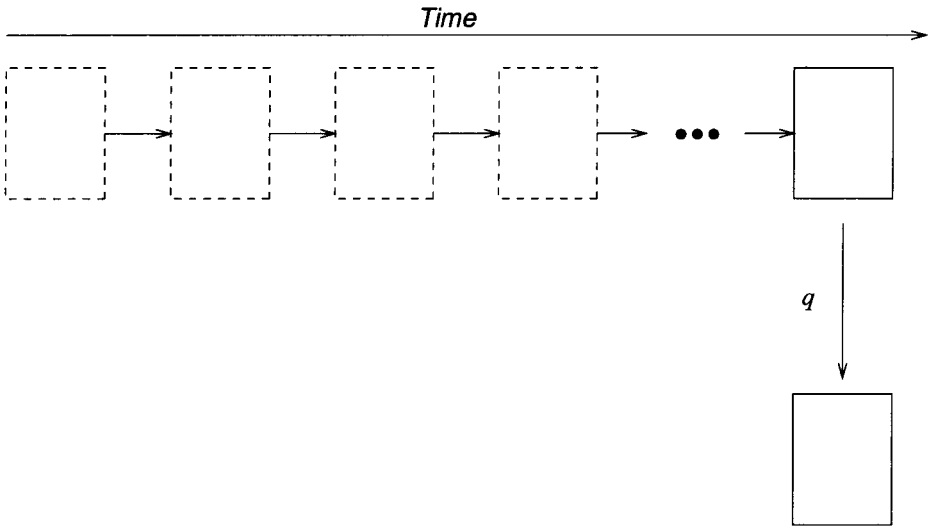

Fig. 1. Level 1 evaluates an SQL3 query over a table without temporal support and returns a table also without temporal support

When a query q is applied to the current state of a table, a resulting table is computed, shown as the rectangle in the bottom right corner. While this figure only concerns queries over single tables, the extension to queries over multiple tables is clear.

Upward compatibility states that (1) all instances of tables in SQL3 are instances of tables in SQL/Temporal, (2) all SQL3 modifications to tables in SQL3 result in the same tables when the modifications are evaluated according to SQL/Temporal semantics, and (3) all SQL3 queries result in the same tables when the queries are evaluated according to SQL/Temporal.

By requiring that SQL/Temporal is a strict superset (i.e., only *adding* language constructs), it is relatively easy to ensure that SQL/Temporal is upward compatible with SQL3.

Throughout, we provide examples of the various levels. In Sec. 5, we show these examples expressed in SQL/Temporal.

EXAMPLE 1: A company wishes to computerize its personnel records, so it creates two tables, an employee table and a monthly salary table. Every employee must have a salary. These tables are populated. A view identifies those

employees with a monthly salary greater than $3500. Then employee Therese is given a 10% raise. Since the salary table has no temporal support, Therese's previous salary is lost. These schema changes and queries can be easily expressed in SQL3. □

4.2 Temporal Upward Compatibility

If an existing or new application needs support for the temporal dimension of the data in one or more tables, the table can be defined with or altered to add valid-time support (e.g., by using the `CREATE TABLE ... AS VALID` or `ALTER ... ADD VALID` statements). The distinction of a table having valid-time support is orthogonal to the many other distinctions already present in SQL/Foundation, including "base table" versus "derived table," "created table" versus "declared table," "global table" versus "local table," "grouped table" versus ungrouped table, ordered table versus table with implementation-dependent order, "subtable" versus "supertable," and "temporary table" versus "permanent table." These distinctions can be combined, subject to stated rules. For example, a table can be simultaneously a temporary table, a table of degree 1, an inherently updatable table, a viewed table and a table with valid-time support. In most of the SQL3 specification, it does not matter what distinctions apply to the table in question. In those few places where it does matter, the syntax and general rules specify the distinction.

It is undesirable to be forced to change the application code that accesses the table without temporal support that is replaced by a table with valid-time support. We formulate a requirement that states that the existing applications on tables without temporal support will continue to work with no changes in functionality when the tables they access are altered to add valid-time support. Specifically, *temporal upward compatibility* requires that each query will return the same result on an associated snapshot database as on the temporal counterpart of the database. Further, this property is not affected by modifications to those tables with valid-time support.

Temporal upward compatibility is illustrated in Fig. 2. When valid-time support is added to a table, the history is preserved, and modifications over time are retained. In this figure, the state to the far left was the current state when the table was made temporal. All subsequent modifications, denoted by the arrows, result in states that are retained, and thus are solid rectangles. Temporal upward compatibility ensures that the states will have identical contents to those states resulting from modifications of the table without valid-time support.

The query q is an SQL3 query. Due to temporal upward compatibility the semantics of this query must not change if it is applied to a table with valid-time support. Hence, the query only applies to the current state, and a table without temporal support results.

EXAMPLE 2: We make both the employee and salary tables temporal. This means that all information currently in the tables is valid from today on. We

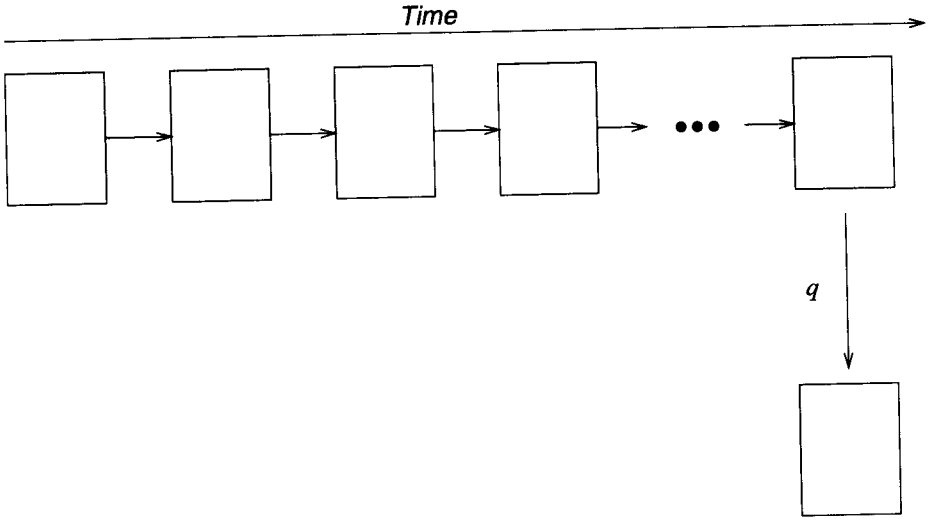

Fig. 2. Level 2 evaluates an SQL3 query over a table with valid-time support and returns a table with similar support

add an employee. This modification to the two tables, consisting of two SQL3 `INSERT` statements, respects temporal upward compatibility. That means it is valid from now on. Queries and views on these tables with newly-added valid-time support work exactly as before. The SQL3 query to list where high-salaried employees live returns the current information. Constraints and assertions also work exactly as before, applying to the current state and checked on database modification. □

It is instructive to consider temporal upward compatibility in more detail. When designing information systems, two general approaches have been advocated. In the first approach, the system design is based on the *function* of the enterprise that the system is intended for (the "Yourdon" approach [19]); in the second, the design is based on the *structure* of the reality that the system is about (the "Jackson" approach [5]). It has been argued that the latter approach is superior because structure may remain stable when the function changes while the opposite is generally not possible. Thus, a more stable system design, needing less maintenance, is achieved when adopting the second design principle. This suggests that the data needs of an enterprise are relatively stable and only change when the actual business of the enterprise changes.

Enterprises currently use non-temporal database systems for database management, but that does not mean that enterprises manage only non-temporal data. Indeed, temporal databases are currently being managed in a wide range of applications, including, e.g., academic, accounting, budgeting, financial, insurance, inventory, legal, medical, payroll, planning, reservation, and scientific

applications. Temporal data may be accommodated by non-temporal database systems in several ways. For example, a pair of explicit time attributes may encode a valid-time interval associated with a row.

Temporal database systems offer increased user-friendliness and productivity, as well as better performance, when managing time-varying data, since they are optimized for such data. The typical situation, when replacing a non-temporal system with a temporal system, is one where the enterprise is not changing its business, but wants the extra support offered by the temporal system for managing its temporal data. Thus, it is atypical for an enterprise to suddenly desire to record temporal information where it previously recorded only snapshot information. Such a change would be motivated by a change in the business.

The typical situation is rather more complicated. The non-temporal database system is likely to already manage temporal data, which is encoded using tables without temporal support, in an ad hoc manner. When adopting the new system, upward compatibility guarantees that it is not necessary to change the database schema or application programs. However, without changes, the benefits of the added valid-time support are also limited. Only when defining new tables or modifying existing applications, can the new temporal support be exploited. The enterprise then gradually benefits from the temporal support available in the system.

Nevertheless, the concept of temporal upward compatibility is still relevant, for several reasons. First, it provides an appealing intuitive notion of a table with valid-time support: the semantics of queries and modification are retained from tables without temporal support; the only difference is that intermediate states are also retained. Second, in those cases where the original table contained no historical information, temporal upward compatibility affords a natural means of migrating to temporal support. In such cases, not a single line of the application need be changed when the table is altered to be temporal. Third, conventional tables that do contain temporal information and for which temporal support has been added can still be queried and modified by conventional SQL3 statements in a consistent manner.

4.3 Sequenced Valid Extensions

The requirements covered so far have been aimed at protecting investments in legacy code and at ensuring uninterrupted operation of existing applications when achieving substantially increased temporal support. Upward compatibility guarantees that (non-historical) legacy application code will continue to work without change when migrating, and temporal upward compatibility in addition allows legacy code to coexist with new temporal applications following the migration.

The requirement in this section aims at protecting the investments in programmer training and at ensuring continued efficient, cost-effective application development upon migration. This is achieved by exploiting the fact that programmers are likely to be comfortable with SQL.

Sequenced valid semantics states that SQL/Temporal must offer, for each query in SQL3, a temporal query that "naturally" generalizes this query, in a specific technical sense. In addition, we require that the SQL/Temporal query be syntactically similar to the SQL3 query that it generalizes.

With this requirement satisfied, SQL3-like SQL/Temporal queries on tables with temporal support have semantics that are easily ("naturally") understood in terms of the semantics of the SQL3 queries on tables without temporal support. The familiarity of the similar syntax and the corresponding, naturally extended semantics makes it possible for programmers to immediately and easily write a wide range of temporal queries, with little need for expensive training.

Fig. 3 illustrates this property. We have already seen that an SQL3 query q on a table with valid-time support applies the standard SQL3 semantics on the current state of that table, resulting in a table without temporal support. This figure illustrates a new query, q', which is an SQL/Temporal query. Query q' is applied to the table with valid-time support (the sequence of states across the top of the figure), and results in a table also with valid-time support, which is the sequence of states across the bottom.

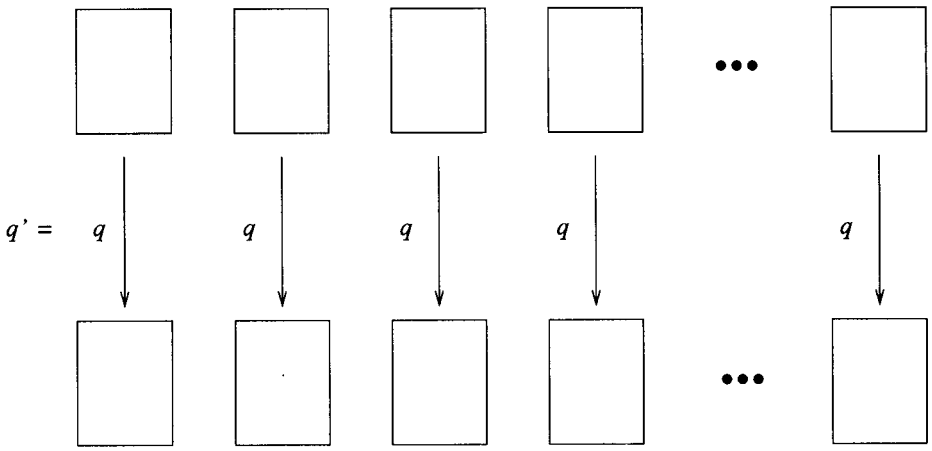

Fig. 3. Level 3 evaluates an SQL/Temporal query over a table with valid-time support and returns a table with similar support

We would like the semantics of q' to be easily understood by the SQL3 programmer. Satisfying sequenced semantics along with the syntactical similarity requirement makes this possible. Specifically, the meaning of q' is precisely that of applying SQL3 query q on each state of the input table (which must have temporal support), producing a state of the output table for each such application. And when q' also closely resembles q syntactically, temporal queries are

easily formulated and understood. To generate query q', one needs only prepend the reserved word VALIDTIME to query q.

EXAMPLE 3: We ask for the history of the monthly salaries paid to employees. Asking that question for the current state (i.e., what is the salary for each employee) is easy in SQL3; let us call this query q. To ask for the history, we simply prepend the keyword VALIDTIME to q to generate the SQL/Temporal query. Sequenced semantics allows us to do this for all SQL3 queries. So let us try a harder one: list *the history of* those employees for which no one makes a higher salary and lives in a different city. Again the problem reduces to expressing the SQL3 query for the current state. We then prepend VALIDTIME to get the history. Sequenced semantics also works for views, integrity constraints and assertions. □

These concepts also apply to sequenced *modifications*, illustrated in Fig. 4. A valid-time modification destructively modifies states as illustrated by the curved arrows. As with queries, the modification is applied on a state-by-state basis. Hence, the semantics of the SQL/Temporal modification is a natural extension of the SQL mφdification statement that it generalizes.

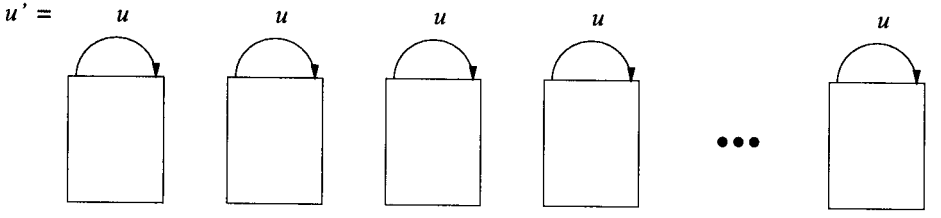

Fig. 4. Level 3 also evaluates an SQL/Temporal modification on a table with valid-time support

EXAMPLE 4: It turns out that a particular employee never worked for the company. That employee is deleted from the database. Note that if we use an SQL3 DELETE statement, temporal upward compatibility requires deleting the information only from the current (and future) states. By prepending the reserved word VALIDTIME to the DELETE statement, we can remove that employee from every state of the table.

Many people misspell the town Tucson as "Tuscon," perhaps because the name derives from an American Indian word in a language no longer spoken. To modify the current state to correct this spelling requires a simple SQL UPDATE statement; let's call this statement u. To correct the spelling in all states, both past and possibly future, we simply prepend the reserved word VALIDTIME to u. □

4.4 Non-Sequenced Queries and Modifications

In a sequenced query, the information in a particular state of the resulting table with valid-time support is derived solely from information in the state at that same time of the source table(s). However, there are many reasonable queries that require other states to be examined. Such queries are illustrated in Fig. 5, in which each state of the resulting table requires information from possibly all states of the source table.

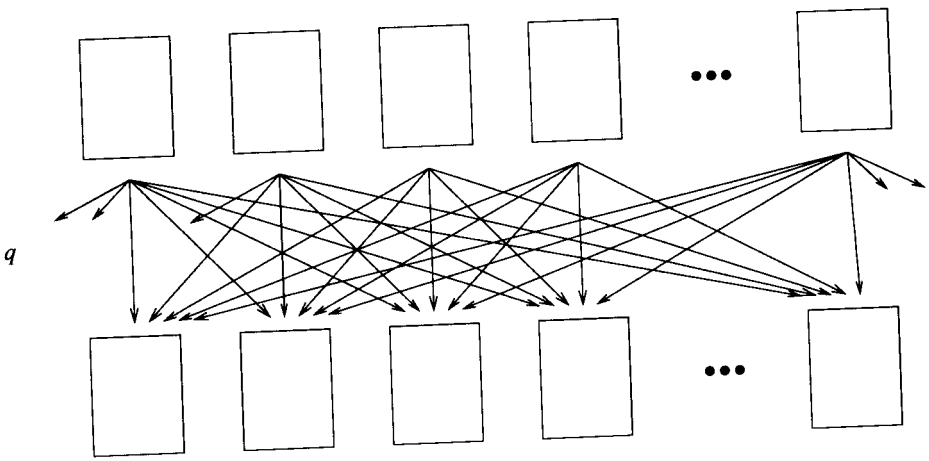

Fig. 5. Level 4 evaluates a non-sequenced SQL/ Temporal query over a table with valid-time support and returns a table with similar support

In this figure, two tables with valid-time support are shown, one consisting of the states across the top of the figure, and the other, the result of the query, consisting of the states across the bottom of the figure. A single query q performs the possibly complex computation, with the information usage illustrated by the downward pointing arrows. Whenever the computation of a single state of the result table may utilize information from a state at a different time, that query is non-sequenced. Such queries are more complex than sequenced queries, and they require new constructs in the query language.

EXAMPLE 5: The query "Who was given salary raises?" requires locating two consecutive times, in which the salary of the latter time was greater than the salary of the former time, for the same employee. Hence, it is a non-sequenced query. □

The concept of non-sequenced queries naturally generalizes to modifications. *Non-sequenced modifications* destructively change states, with information retrieved from possibly all states of the original table. In Fig. 6, each state of the table with valid-time support is possibly modified, using information from possi-

bly all states of the table before the modification. Non-sequenced modifications include future modifications.

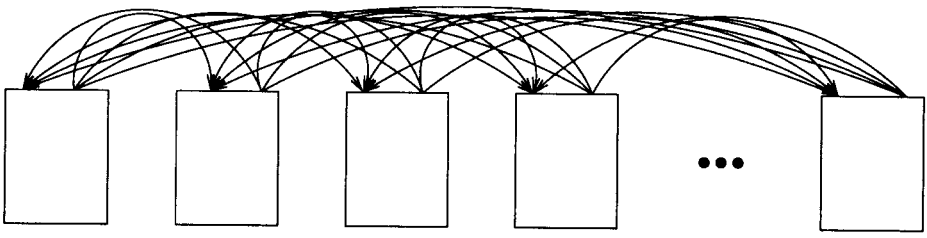

Fig. 6. Level 4 also evaluates a non-sequenced SQL/Temporal modification on a table with valid-time support

EXAMPLE 6: We wish to give employees a 5% raise if they have never had a raise before. This is not a temporally upward compatible modification, because the modification of the current state uses information in the past. For the same reason, it is not a sequenced update. So we must use a slightly more involved SQL/Temporal `UPDATE` statement. In fact, only the predicate "if they never had a raise" need be nonsequenced; the rest of the update can be temporally upward compatible. □

Views and cursors can also be nonsequenced.

EXAMPLE 7: We wish to define a snapshot view of the `salary` table in which the row's timestamp period appears as an explicit column. We can also define a valid-time view on this snapshot view that uses the explicit period column as an implicit timestamp. □

It is important to note that nonsequenced queries are very different from sequenced queries. In the latter, the query language is providing a temporal semantics; in the former, the query language interprets the timestamp as simply another column. For the user, this means that in nonsequenced queries (modifications, assertions, etc.) the period timestamps must be manipulated explicitly. The operations, such as join and relational difference, are performed with respect to the periods themselves, rather than on the individual states of the tables with temporal support. Reserved words are used to syntactically differentiate temporally upward compatible queries, sequenced queries, and non-sequenced queries, each of which applies a distinct semantics.

4.5 Summary

In this section, we have formulated three important requirements that SQL/Temporal should satisfy to ensure a smooth transition of legacy application code. We review each in turn.

Upward compatibility and temporal upward compatibility guarantee that legacy application code needs no modification when migrating and that new temporal applications may coexist with existing applications. They are thus aimed at protecting investments in legacy application code.

The requirement that there be a sequenced temporal extension of all existing statements ensures that the extended query language is easy to use for programmers familiar with the existing query language. The requirement thus helps protect investment in programmer training. It also turns out that this property makes the semantics of tables with valid-time support straight-forward to specify and enables a wide range of implementation alternatives [14].

These requirements induce four levels of temporal functionality, to be defined in SQL/Temporal.

Level 1 This lowest level captures the minimum functionality necessary for the query language to satisfy upward compatibility with SQL3. Thus, there is support for legacy SQL3 statements, but there are no tables with valid-time support and no temporal queries. Put differently, the functionality at this level is identical to that of SQL3.

Level 2 This level adds to the previous level solely by allowing for the presence of tables with valid-time support. The temporal upward compatibility requirement is applicable to this subset of SQL/Temporal. This level adds no new syntax for queries or modifications—only queries and modifications with SQL3 syntax are possible.

Level 3 The functionality of Level 2 is enhanced with the possibility of giving sequenced temporal functionality to queries, views, constraints, assertions, and modifications on tables with valid-time support. This level of functionality is expected to provide adequate support for many applications. Starting at this level, temporal queries exist, so SQL/Temporal must be a sequenced-consistent extension of SQL3.

Level 4 Finally, the full temporal functionality normally associated with a temporal language is added, specifically, non-sequenced temporal queries, assertions, constraints, views, and modifications. These additions include temporal queries and modifications that have no syntactic counterpart in SQL3.

5 Tables with Valid-Time Support in SQL3

This section informally introduces the new constructs of SQL/Temporal. These constructs are an improved and extended version of those in the consensus temporal query language TSQL2 [13]. The improvements concern guaranteeing the properties listed in Sec. 4, to support easy migration of legacy SQL3 application code [2]. The extensions concern views, assertions, and constraints (specifically

temporal upward compatible and sequenced and non-sequenced extensions) that were not considered in the original TSQL2 design.

The presentation is divided into four levels, where each successive level adds temporal functionality. The levels correspond to those discussed informally in the previous section. Throughout, the functionality is exemplified with input to and corresponding output from a prototype system [16]. The reader may find it instructive to execute the sample statements on the prototype.

5.1 Level 1: Upward Compatibility

Level 1 ensures upward compatibility (see Fig. 1), i.e., it guarantees that legacy SQL3 statements evaluated over databases without temporal support return the result dictated by SQL3.

SQL3 Extensions Obviously there are no syntactic extensions to SQL3 at this level.

A Quick Tour The following statements are executed on January 1, 1995. A company creates two tables, an employee table and a monthly salary table. Every employee must have a salary. These schema changes can be easily expressed in SQL3.

```
CREATE TABLE employee(ename VARCHAR(12), eno INTEGER PRIMARY KEY,
              street VARCHAR(22), city VARCHAR(10), birthday DATE)
CREATE TABLE salary(eno INTEGER PRIMARY KEY REFERENCES employee,
              amount INTEGER)

CREATE ASSERTION emp_has_sal CHECK
(NOT EXISTS ( SELECT *
              FROM employee AS e
              WHERE NOT EXISTS ( SELECT *
                                 FROM salary AS s
                                 WHERE e.eno = s.eno)))
```

These tables are populated.

```
INSERT INTO employee
        VALUES ('Therese', 5873, 'Bahnhofstrasse 121', 'Zurich',
                DATE '1961-03-21')
INSERT INTO employee
        VALUES ('Franziska', 6542, 'Rennweg 683', 'Zurich',
                DATE '1963-07-04')

INSERT INTO salary VALUES (6542, 3200)
INSERT INTO salary VALUES (5873, 3300)
```

A view identifies those employees with a monthly salary greater than $3500.

```
CREATE VIEW high_salary AS SELECT * FROM salary WHERE amount > 3500
```

Employee Therese is given a 10% raise. Since the salary table has no temporal support, Therese's previous salary is lost.

```
UPDATE salary s
SET amount = 1.1 * amount
WHERE s.eno = (SELECT e.eno
               FROM employee e
               WHERE e.ename = 'Therese')

COMMIT
```

5.2 Level 2: Temporal Upward Compatibility

Level 2 ensures temporal upward compatibility as depicted in Fig. 2. Temporal upward compatibility is straightforward for queries. They are evaluated over the current state of a database with valid-time support.

SQL3 Extensions The create table statement is extended to define tables with valid-time support. Specifically, this statement can be followed by the clause "AS VALIDTIME <datetime field>", e.g., "AS VALIDTIME PERIOD(DATE)." This specifies that the table has valid-time support, with states indexed by particular days. The alter table statement is extended to permit valid-time support to be added to a table without such support or dropped from a table with valid-time support.

A table with valid-time support is conceptually a sequence of states indexed with valid-time granules at the specified granularity. This is the view of a table with valid-time support adopted in temporal upward compatibility and sequenced semantics. At a more specific logical level, a table with valid-time support is *also* a collection of rows associated with valid-time periods.

Indeed, our definition of the semantics of the addition to SQL/Temporal being proposed satisfies temporal upward compatibility and sequenced semantics.

Quick Tour: Part 2 The following statements are executed on February 1, 1995.

```
ALTER TABLE salary ADD VALIDTIME PERIOD(DATE)
ALTER TABLE employee ADD VALIDTIME PERIOD(DATE)
```

The following statements are typed in the next day (February 2, 1995).

```
INSERT INTO employee
VALUES('Lilian', 3463, '46 Speedway', 'Tuscon',
               DATE '1970-03-09')
INSERT INTO salary VALUES(3463, 3400)
COMMIT
```

The `employee` table contains the following rows. (In these examples, we used open-closed ("[...)") for periods.)

ename	eno	street	city	birthday	Valid
Therese	5873	Bahnhofstrasse 121	Zurich	1961-03-21	[1995-02-01 - 9999-12-31)
Franziska	6542	Rennweg 683	Zurich	1963-07-04	[1995-02-01 - 9999-12-31)
Lilian	3463	46 Speedway	Tuscon	1970-03-09	[1995-02-02 - 9999-12-31)

Note that the valid time extends to the *end of time*, which in SQL3 is the largest date.

The `salary` table contains the following rows.

eno	amount	Valid
6542	3200	[1995-02-01 - 9999-12-31)
5873	3630	[1995-02-01 - 9999-12-31)
3463	3400	[1995-02-02 - 9999-12-31)

We continue, still on February 2. Tables, views, and queries act like before, because temporal upward compatibility is satisfied. To find out where the high-salaried employees live, use the following.

```
SELECT ename, city
FROM   high_salary AS s, employee AS e
WHERE  s.eno = e.eno
```

Evaluated over the current state, this returns the employee Therese, in Zürich.

Assertions and referential integrity act like before, applying to the current state. The following transaction will abort due to (1) a violation of the `PRIMARY KEY` constraint, (2) a violation of the `emp_has_sal` assertion and (3) a referential integrity violation, respectively.

```
INSERT INTO employee
        VALUES ('Eric', 3463, '701 Broadway', 'Tucson',
                DATE '1988-01-06')
INSERT INTO employee
        VALUES ('Melanie', 1234, '701 Broadway', 'Tucson',
                DATE '1991-03-08')
INSERT INTO salary VALUES(9999, 4900)
COMMIT
```

5.3 Level 3: Sequenced Language Constructs

Level 3 adds syntactically similar, sequenced counterparts of existing queries, modifications, views, constraints, and assertions (see Fig. 3). Sequenced SQL/Temporal queries produce tables with valid-time support. The state of a result table at each time is computed from the state of the underlying table(s) at the

same time, via the semantics of the contained SQL3 query. In this way, users are able to express temporal queries in a natural fashion, exploiting their knowledge of SQL3. Temporal views, assertions and constrains can likewise be naturally expressed.

SQL3 Extensions Temporal queries, modifications, views, assertions, and constraints are signaled by the reserved word `VALIDTIME`. This reserved word can appear in a number of locations.

Derived table in a from clause In the from clause, one can prepend `VALIDTIME` to a <query expression>.

View definition Temporal views can be specified, with sequenced semantics.

Assertion definition A sequenced assertion applies to each of the states of the underlying table(s). This is in contrast to a snapshot assertion, which is only evaluated on the current state. In both cases, the assertion is checked before a transaction is committed.

Table and column constraints When specified with `VALIDTIME`, such constraints must apply to each state of the table with valid-time support.

Cursor expression Cursors can range over tables with valid-time support.

Single-row select Such a select can return a row with an associated valid time.

Modification statements When specified with `VALIDTIME`, the modification applies to each state comprising the table with valid-time support.

In all cases, the `VALIDTIME` reserved word indicates that sequenced semantics is to be employed.

An optional period expression after `VALIDTIME` specifies that the valid-time period of each row of the result is intersected with the value of the expression. This allows one to restrict the result of a select statement, cursor expression, or view definition to a specified period, and to restrict the time for which assertion definitions, table constraints and column constraints are checked.

Quick Tour: Part 3 We evaluate the following statements on March 1, 1995.

Prepending `VALIDTIME` to any `SELECT` statement evaluates that query on all states, in a sequenced fashion. The first query provides the history of the monthly salaries paid to employees. This query is constructed by first writing the snapshot query, then prepending `VALIDTIME`.

```
VALIDTIME SELECT ename, amount
FROM salary AS s, employee AS e
WHERE s.eno = e.eno
```

This evaluates to the following.

ename	amount	Valid
Franziska	3200	[1995-02-01 - 9999-12-31)
Therese	3630	[1995-02-01 - 9999-12-31)
Lilian	3400	[1995-02-02 - 9999-12-31)

List those for which no one makes a higher salary in a different city, over all time.

```
VALIDTIME SELECT ename
FROM employee AS e1, salary AS s1
WHERE e1.eno = s1.eno
   AND NOT EXISTS (SELECT ename
                   FROM employee AS e2, salary AS s2
                   WHERE e2.eno = s2.eno
                      AND s2.amount > s1.amount
                      AND e1.city <> e2.city)
```

This gives the following result.

ename	Valid
Therese	[1995-02-01 - 9999-12-31)
Franziska	[1995-02-01 - 1995-02-02)

Therese is listed because the only person in a different city, Lilian, makes a lower salary. Franziska is listed because for that one day, there was no one in a different city (Lilian did not join the company until February 2).

The reserved word VALIDTIME specifies that the semantics of the query to which it is prepended is a sequenced semantics. Conceptually the query is evaluated independently on every state of the underlying tables (cf. Fig. 3). This ensures that the user's intuition about SQL carries over to sequenced queries and modifications.

A formal semantics for sequenced queries has been developed [14, 3]. While Fig. 3 provides the meaning of sequenced queries in terms of *states*, the formal semantics is expressed in terms of manipulations on the period timestamps of the underlying tables with valid-time support.

We then create a temporal view, similar to the non-temporal view defined earlier. In fact, the only difference is the use of the reserved word VALIDTIME.

```
CREATE VIEW high_salary_history AS
VALIDTIME SELECT * FROM salary WHERE s.salary > 3500
```

Finally, we define a temporal column constraint.

```
ALTER TABLE salary
ADD VALIDTIME CHECK (amount > 1000 AND amount < 12000)
COMMIT
```

Rather than being checked on the current state only, this constraint is checked on each state of the salary table. This is useful to restrict *retroactive* changes [6], i.e., changes to past states and *predictive* changes, i.e., changes to future states. This constraint is satisfied for all states in the table.

Sequenced modifications are similarly handled. To remove employee number 5873 for all states of the database, we use the following statement.

```
VALIDTIME DELETE FROM employee WHERE eno = 5873
VALIDTIME DELETE FROM salary WHERE eno = 5873
COMMIT
```

To correct the common misspelling of Tucson, we use the following statement.

```
VALIDTIME UPDATE employee
SET city = 'Tucson'
WHERE city = 'Tuscon'
COMMIT
```

This updates all incorrect values, at all times, including the past and future. Lillian's city is thus corrected.

5.4 Level 4: Non-Sequenced Language Constructs

Level 4 accounts for non-sequenced queries (see Fig. 5) and non-sequenced modifications (see Fig. 6). Many useful queries and modifications are in this category. However, their semantics is necessarily more complicated than that of sequenced queries, because non-sequenced queries cannot exploit that useful property. Instead, they must support the formulation of special-purpose user-defined temporal relationships between implicit timestamps, datetime values expressed in the query, and stored datetime columns in the database.

Nonsequenced SQL/Temporal queries can produce tables with or without valid-time support, depending on whether the valid-time period of the resulting rows is provided in the query. The state of a result table, if a table is without valid-time support, or the state of a result table at each time, if a table has valid-time support, is computed from potentially all of the states of the underlying table(s), at any time. The semantics are quite simple. A nonsequenced evaluation treats a table with valid-time support as a table without temporal support, but with an additional column containing the timestamp. We again emphasize that this semantics is quite different from temporally upward compatible semantics (where the query is evaluated only on the current state) and from sequenced semantics (where the query is effectively evaluated on each state independently).

SQL3 Extensions Nonsequenced valid queries are signaled by the new reserved word `NONSEQUENCED` preceding the reserved word `VALIDTIME`. This applies analogously to nonsequenced modifications, views, assertions, and constraints. This reserved word can appear in a number of locations.

Derived table in a from clause In the from clause, one can prepend `NONSEQUENCED VALIDTIME` to a <query expression>. This results in a table without temporal support, and is the means of removing the valid-time support of a table.

View definition Nonsequenced views can be specified.

Assertion definition A nonsequenced assertion applies to the underlying table(s), considered as snapshot tables with an additional explicit timestamp column. This is in contrast to a snapshot assertion, which is only evaluated on the current state. In both cases, the assertion is checked before a transaction is committed.

Table and column constraints When specified with `NONSEQUENCED VALIDTIME`, such constraints apply to the table with the valid timestamp treated as an explicit column.

Cursor expression Cursors can range over the result of a nonsequenced select.

Single-row select A nonsequenced single-row select will return a row without temporal support, even when evaluated over tables with valid-time support.

Modification statements When specified with `NONSEQUENCED VALIDTIME`, the modification applies to the table considered as a snapshot table.

In all cases, the `NONSEQUENCED` reserved word indicates that nonsequenced semantics is to be employed.

The syntax of a <query expression> is extended to the following.

{ { `NONSEQUENCED` } `VALIDTIME` { <value expression> } }
<query expression>

An optional period expression after `NONSEQUENCED VALIDTIME` specifies the valid-time period of each row of the result, and thus renders the resulting table to have valid-time support. This enables a table without temporal support to be converted into a table with valid-time support within a query or other statement.

For modification statements, the period expression after `VALIDTIME` specifies the temporal scope of the modification: the times at which the modification is to be applied.

The value expression "`VALIDTIME`(<correlation name>)" is available; it evaluates to the valid-time period of the row associated with the correlation or table name. This is required because valid-time periods of tables with valid-time support are not explicit columns (the alternative violates temporal upward compatibility).

The following quick tour provides examples of these constructs.

Quick Tour: Part 4 This quick tour starts with the database as it was when we last left it, in the previous quick tour. The `employee` table has the following contents.

ename	eno	street	city	birthday	Valid
Franziska	6542	Rennweg 683	Zurich	1963-07-04	[1995-02-01 - 9999-12-31)
Lilian	3463	46 Speedway	Tucson	1970-03-09	[1995-02-02 - 9999-12-31)

The salary table has the following contents.

eno	amount	Valid
6542	3200	[1995-02-01 - 9999-12-31)
3463	3400	[1995-02-02 - 9999-12-31)

A period expression after VALIDTIME specifies the temporal scope of the result. List those who were employed sometime during the first six months.

```
VALIDTIME PERIOD '[1995-01-01 - 1995-07-01)'
SELECT ename FROM employee
```

This returns the following table.

ename	Valid
Franziska	[1995-02-01 - 1995-07-01)
Lilian	[1995-02-02 - 1995-07-01)

On April 1, 1995, we give Lilian a 5% raise, starting immediately. This is a temporally upward compatible modification, and so is already expressible in SQL.

```
UPDATE salary
SET amount = 1.05 * amount
WHERE eno = (SELECT S.eno
             FROM salary AS S, employee as E
             WHERE ename = 'Lilian' AND E.eno = S.eno)
COMMIT
```

This results in the following salary table.

eno	amount	Valid
6542	3200	[1995-02-01 - 9999-12-31)
3463	3400	[1995-02-02 - 1995-04-01)
3463	3570	[1995-04-01 - 9999-12-31)

To determine who was given salary *raises*, we must simultaneously consider two consecutive states of the salary table, before and after the raise. This requires a nonsequenced query.

```
NONSEQUENCED VALIDTIME SELECT ename
FROM employee AS E, salary AS S1, salary AS S2
WHERE E.eno = S1.eno AND E.eno = S2.eno
   AND S1.amount < S2.amount
   AND VALIDTIME(S1) MEETS VALIDTIME(S2)
```

MEETS ensures that the valid-time period associated with S1 is immediately followed by the valid-time period associated with S2. Since the valid-time period of a row is not in an explicit column (as this would violate temporal upward compatibility), VALIDTIME() is used to extract the associated valid-time period. The result is a table without temporal support, because NONSEQUENCED is not followed by a period expression.

ename
Lilian

If we instead wish to get back a table with valid-time support, i.e., "Who was given salary raises, and when did they receive the higher salary?", we place a <value expression> after VALIDTIME to specify when each resulting row is valid. Our first try is the following, in which the <value expression> extracts the valid timestamp of S2.

```
NONSEQUENCED VALIDTIME VALIDTIME(S2) SELECT ename
FROM employee AS E, salary AS S1, salary AS S2
WHERE E.eno = S1.eno AND E.eno = S2.eno
   AND S1.amount < S2.amount
   AND VALIDTIME(S1) MEETS VALIDTIME(S2)
```

Because an expression is associated with NONSEQUENCED VALIDTIME, the result will be a table with valid-time support, with a valid timestamp of the value of the timestamp of S2. However, this is not quite correct, because the period expression following VALIDTIME can only mention the columns of the following select statement, and the timestamp of S2 is not available. So we put the value in the select list and use an enclosing (sequenced) select statement to get rid of this extra column.

```
VALIDTIME SELECT ename
FROM (NONSEQUENCED VALIDTIME S2valid SELECT ename,
                                    VALIDTIME(S2) AS S2valid
      FROM employee AS E, salary AS S1, salary AS S2
      WHERE E.eno = S1.eno AND E.eno = S2.eno
         AND S1.amount < S2.amount
         AND VALIDTIME(S1) MEETS VALIDTIME(S2) ) AS S
```

The inner query evaluates to two columns, ename and S2valid. The NONSEQUENCED VALIDTIME includes a <value expression>, specifying that a table with valid-time support is desired. The valid timestamp of each row is the same as the value of the S2valid column. The outer query just projects out the ename column, retaining the valid timestamp. This query has the following result.

ename	Valid
Lilian	[1995-04-01 - 9999-12-31)

If we had desired the time when the person had received the *lower* salary, we would simply specify `VALIDTIME(S1)` instead.

This query is admittedly more complex to specify than the sequenced queries given in the previous section. In nonsequenced queries the user is doing all the work of manipulating the timestamps, whereas in sequenced queries, the DBMS handles the timestamps automatically, freeing the user from this concern. The reason that nonsequenced queries are included is that some (very useful) queries cannot be expressed using the sequenced semantics, the query just given being one example.

Following `VALIDTIME` with a period expression in a modification (whether sequenced or not) specifies the temporal scope of the modification. Two applications of this are retroactive and future changes. Assume it is now May 1, 1995. Franziska, employee 6542, will be taking a leave of absence the last half of the year.

```
VALIDTIME PERIOD '[1995-07-01 - 1996-01-01)'
DELETE FROM salary
WHERE eno = 6542

VALIDTIME PERIOD '[1995-07-01 - 1996-01-01)'
DELETE FROM employee
WHERE eno = 6542

COMMIT
```

The `salary` table now has the following contents.

eno	amount	Valid
6542	3200	[1995-02-01 - 1995-07-01)
6542	3200	[1996-01-01 - 9999-12-31)
3463	3400	[1995-02-02 - 1995-04-01)
3463	3570	[1995-04-01 - 9999-12-31)

The `employee` table has the following contents.

ename	eno	street	city	birthday	Valid
Franziska	6542	Rennweg 683	Zurich	1963-07-04	[1995-02-01 - 1995-07-01)
Franziska	6542	Rennweg 683	Zurich	1963-07-04	[1996-01-01 - 9999-12-31)
Lilian	3463	46 Speedway	Tucson	1970-03-09	[1995-02-02 - 9999-12-31)

Note that these deletions split single periods into two, with a lapse between them. Many modifications are greatly simplified in this way. Also note that previously specified sequenced valid referential integrity and other constraints and assertions must apply to each state. Hence, if the first `DELETE` was performed, but not the second, the `COMMIT` will abort because the `emp_has_sal` constraint is violated for certain states, such as the one on August 1, 1995.

The period expression following `VALIDTIME` is also allowed for assertions and constraints. Assume that no employee may make less than 3000 during 1996.

```
CREATE ASSERTION salary_check
VALIDTIME PERIOD '[1996-01-01 - 1997-01-01)' CHECK
   (NOT EXISTS ( SELECT * FROM salary WHERE amount < 3000 ) )
```

This is a sequenced assertion, and thus applies separately to each state in 1996. Nonsequenced assertions and constraints apply to all states at once. To assert that there is only one employee with a particular name, we use the following constraint within the `employee` table definition.

```
CONSTRAINT unique_name UNIQUE (ename)
```

This is interpreted with temporal upward compatible semantics, and so applies only to the current state. If all we do is temporal upward compatible modifications, this will be sufficient. However, if we perform future updates, violations may be missed. To always check all states, a sequenced constraint is used.

```
CONSTRAINT unique_name_per_time VALIDTIME UNIQUE (ename)
```

This will ensure that at any time, each ename value is unique.

To ensure that each ename is unique, *across all states simultaneously*, a nonsequenced constraint is required.

```
CONSTRAINT unique_name_over_all_time
NONSEQUENCED VALIDTIME UNIQUE (ename)
```

The above `employee` table satisfies the first two constraints, but not the third (the nonsequenced one), because there are two rows with an ename of Franziska.

As with `VALIDTIME`, `NONSEQUENCED VALIDTIME` can appear in a from clause. To give employees a 5% raise if they never had a raise before, we first write a temporal upward compatible modification (i.e., without `VALIDTIME`) to give the raise.

```
UPDATE salary AS S
SET amount = 1.05 * amount
```

We can augment this statement to use a non-sequenced query in the from clause to look for raises in the past.

```
UPDATE salary AS S
SET amount = 1.05 * amount
WHERE NOT EXISTS
          (SELECT *
           FROM (NONSEQUENCED VALIDTIME SELECT *
                 FROM salary AS S1, salary AS S2
                    WHERE S1.amount < S2.amount
                       AND VALIDTIME(S1) MEETS VALIDTIME(S2)
                       AND S1.eno = S.eno) AS S3
           )
    AND S.eno = S3.eno
COMMIT
```

The NOT EXISTS was added. Assume that the update was entered on June 1, 1995. The following salary table results.

eno	amount	Valid
6542	3200	[1995-02-01 - 1995-06-01)
6542	3360	[1995-06-01 - 1995-07-01)
6542	3360	[1996-01-01 - 9999-12-31)
3463	3400	[1995-02-02 - 1995-04-01)
3463	3570	[1995-04-01 - 9999-12-31)

Since the update is evaluated with temporal upward compatible semantics, it changes the salary for valid times after June 1.

Finally, we wish to define a snapshot view of the salary table in which the row's timestamp appears as an explicit column, here when.

```
CREATE VIEW snapshot_salary (eno, amount, when) AS
NONSEQUENCED VALIDTIME SELECT S.*, VALIDTIME(S) FROM salary AS S
```

Coming around full circle, we can define a valid-time view on snapshot_salary that uses the explicit column validtime as an implicit timestamp.

```
CREATE VIEW temporal_salary (eno, amount) AS
VALIDTIME SELECT eno, amount
FROM (NONSEQUENCED VALIDTIME when SELECT *
      FROM snapshot_salary AS S) AS S2
```

This conversion can also be applied within queries and cursors.

6 Transaction-Time Support

Transaction time identifies when data was asserted in the database. If transaction time is supported, the states of the database at all previous points of time are retained and modifications are append-only.

Unlike valid time, transaction time cannot be entirely simulated with tables with explicit timestamp columns. The reason is that tables with transaction-time support are *append-only*: they grow monotonically. Specifically, while the query functionality can be simulated on tables with no temporal support, in the same way that valid-time query functionality can be translated into queries on tables with no temporal support, there is no way to restrict the user to modifications that ensure the table is append-only. While one can revoke permission to use DELETE, it is still possible for the user to corrupt the transaction timestamp via database updates and insertions. This means that the user can never be sure that what the table says was stored at some time in the past was actually in the table at that time. The only way to ensure the consistency of the data is to have the DBMS maintain the transaction timestamps automatically.

Many applications need to keep track of the past states of the database, often for audit traceability requirements. Changes are not allowed on the past states;

that would prevent secure auditing. Instead, compensating transactions are used to correct errors.

When an error is encountered, often the analyst will look at the state of the database at a previous point in time to determine where and how the error occurred.

However, SQL-92 (nor the current SQL3 draft) does not support such modifications or queries well. The following example will illustrate the problems.

- Assume that we wish to keep track of the changes and deletions of the `employee` table. If standard SQL was used, this table would have six columns: `ename`, `eno`, `street`, `city`, `birthdate`, and `When` (a `PERIOD` indicating when the row was valid). To know when rows are inserted and (logically) deleted, we add two more columns, `InsertTime` and `DeleteTime`, both of the data type `TIMESTAMP`. Of course, adding these two columns breaks the referential integrity constraint between `salary.eno` and `employee.eno`. The reader is invited to write this referential integrity constraint to take into account the three time columns.
- We ask "How many highly paid employees have been in each city?" This query is quite complex to formulate in SQL.
- It turns out that one of the cities shows an unreasonable number of highly-paid current employees (more than 25). When was the error introduced? Is this inconsistency in the database widespread? How long has the database been incorrect? The query "When did we think that there were many highly-paid employees in Tuscon?" provides an initial answer, but is also very difficult to express in SQL.

These queries are very challenging, even for SQL experts, when time is involved.

Modifications are even more of a problem. A logical deletion must be implemented as an update and an insertion, because we do not want to change the previously stored information. However, there is no way of preventing an application from inadvertently corrupting past states (by incorrectly altering the values of the InsertTime or DeleteTime columns), or a white-collar criminal from intentionally "changing history" to cover up his tracks.

The solution is to have the DBMS maintain transaction time automatically, so that the integrity of the previous states of the database is preserved. The query language can also help out, by making it easy to write queries and modifications.

With the small syntactic additions proposed here, transaction time can be easily added.

```
ALTER TABLE employee ADD TRANSACTIONTIME
```

Because the DBMS is maintaining transaction time for us, for this table, we do not have to worry about the integrity of the previous states. The DBMS simply would not let us modify past states.

The previously specified sequenced valid referential integrity still applies, always on the current state of the database. No rephrasing of this integrity constraint is necessary.

The query "How many highly paid employees have been in each city?" asks for the history in valid time of the current transaction-time state. Hence, it is particularly easy to specify, by exploiting transaction-time upward compatibility.

```
VALIDTIME SELECT city, COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
GROUP BY city
```

To find where the error was made, we write the query "When did we think that there are many highly-paid employees in Tucson?" This uses the current time in valid time ("are"), but looks at past states of the database ("when did we think"). This requires a sequenced transaction query, with valid-time upward compatibility.

```
TRANSACTIONTIME SELECT COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
        AND city = 'Tucson'
GROUP BY city
HAVING COUNT(*) > 25
```

By having the DBMS maintain transaction time, applications that need to retain past states of tables for auditing purposes can have these past states maintained automatically, correctly, and securely. As well, the proposed language extensions enable queries to be written in minutes instead of hours.

The concepts of temporal upward compatibility *(TUC)*, sequenced *(SEQ)*, and nonsequenced *(NONSEQ)* semantics apply orthogonally to valid time and transaction time.

The semantics is dictated by three simple rules.

- The absence of `VALIDTIME` (respectively, `TRANSACTIONTIME`) indicates valid-time (resp., transaction-time) upward compatibility. The result does not include valid-time (resp., transaction-time) support.
- `VALIDTIME` (respectively, `TRANSACTIONTIME`) indicates sequenced valid (resp., transaction) semantics. An optional period expression temporally scopes the result. The result includes valid-time (resp., transaction-time) support.
- `NONSEQUENCED` denotes nonsequenced valid (resp., transaction) semantics. An optional period expression after `NONSEQUENCED VALIDTIME` provides a valid-time timestamp, yielding valid-time support in the result.

EXAMPLE 8: Starting with the simple query "Which Tucson employees are paid highly?" we can state queries that are different combinations of *TUC*, *SEQ*, and *NONSEQ* in valid and transaction time. In the following, we indicate valid time, then transaction time. Hence, "*TUC/SEQ*" means valid-time upward compatible and sequenced transaction-time semantics.

TUC/TUC Which Tucson employees are current paid highly?
A table with no temporal support results.

SEQ/TUC Which Tucson employees are or were paid highly (as best known)?
Note the the employee had to be in Tucson at the same time they were highly paid. A table with valid-time support results.

TUC/SEQ Who did we think are the highly-paid Tucson employees?
A table with transaction-time support results.

NONSEQ/TUC Which highly-paid employees lived at some time in Tucson, as best known?
A table with no temporal support results.

TUC/NONSEQ When was it recorded that a Tucson employee is currently paid highly?
A table with no temporal support results.

SEQ/SEQ When did we think that some Tucson employee was paid highly, at the same time?
A table with both valid-time and transaction-time support results.

SEQ/NONSEQ When did we correct the information to record that some Tucson employee was paid highly?
A table with valid-time support results. For each transaction time, we get a row with valid-time support, indicating when the employee is now considered to be in Tucson and be highly paid.

NONSEQ/SEQ Who was recorded, perhaps erroneously, to have resided in Tucson at some time and was paid highly, perhaps at some other time?
Here we get a table with transaction-time support, indicating when the perhaps erroneous data was in the table.

NONSEQ/NONSEQ When did we correct the information, to record that some Tucson employee was paid highly, perhaps at some other time?
Here a table with no temporal support results.

TUC in valid time translates in English to "at now;" *SEQ* translates to "at the same time;" and *NONSEQ* translates to "at any time." *TUC* in transaction time translates to "as best known;" *SEQ* translates to "when did we think ... at the same time;" and *NONSEQ* translates to "when was it recorded that."
This example illustrates that all combinations are meaningful and useful. □

While this example emphasized the orthogonally of valid and transaction time, that *TUC*, *SEQ*, and *NONSEQ* can be applied equally to both, there are still some differences between the two types of time. First, valid time can have a precision specified by the user at table creation time. The transaction timestamps have an implementation-dependent range and precision. Second, valid time extends into the future, whereas transaction time always ends at now. Third, unlike a <value expression> following `NONSEQUENCED VALIDTIME`, a <value expression> is not permitted after `NONSEQUENCED TRANSACTIONTIME`, because it is not possible to compute a transaction timestamp. Such a timestamp may only be inferred via a sequenced transaction query. Finally, during modifications the DBMS provides the transaction time of facts, in contrast with

the valid time, which is provided by the user. This derives from the different semantics of transaction time and valid time. Specifically, when a fact is (logically) deleted from a table with transaction-time support, its transaction stop time is set automatically by the DBMS to the current time. When a fact is inserted into the table, its transaction start time is set by the DBMS, again to the current time. An update is treated, concerning the transaction timestamps, as a deletion followed by an insertion. The transaction times that a set of modification transactions give to the modified rows must be consistent with the serialization order of those transactions.

The following examples will emphasize the parallel between valid-time and transaction-time support. Specifically, temporal upward compatibility guarantees that conventional, nontemporal queries, updates, etc. work as before, with the same semantics. Since the history of the database is recorded in tables with both valid-time and transaction-time support, we can find out when corrections were made, using a nonsequenced transaction query. Modifications take effect at the current transaction time. However, we can still specify the scope of the change in valid time, both before and after now (retroactive and postactive changes, respectively). Finally, arbitrarily complex queries in transaction time can be expressed with nonsequenced transaction queries. As always, the concepts also apply to views, cursors, constraints, and assertions.

Quick Tour: Part 5 This quick tour starts with the database as it was when we last left it, at the end of the previous quick tour. The `employee` table has the following contents. Recall that closed-open periods are used here for the valid-time and transaction-time periods.

ename	eno	street	city	birthday	Valid
Franziska	6542	Rennweg 683	Zurich	1963-07-04	[1995-02-01 - 1995-07-01)
Franziska	6542	Rennweg 683	Zurich	1963-07-04	[1996-01-01 - 9999-12-31)
Lilian	3463	46 Speedway	Tucson	1970-03-09	[1995-02-02 - 9999-12-31)

The `salary` table has the following contents.

eno	amount	Valid
6542	3200	[1995-02-01 - 1995-06-01)
6542	3360	[1995-06-01 - 1995-07-01)
6542	3360	[1996-01-01 - 9999-12-31)
3463	3400	[1995-02-02 - 1995-04-01)
3463	3570	[1995-04-01 - 9999-12-31)

We can alter the `employee` table to be a table with both valid-time and transaction-time support, by adding transaction-time support. Assume that the current date is July 1, 1995.

```
ALTER TABLE employee ADD TRANSACTIONTIME
COMMIT
```

Since **employee** was a table with valid-time support, this statement converts it to the following table with both valid-time and transaction-time support. Recall that an the ending bound of the transaction-time period equal to the end of time simply indicates that the row still logically resides in the table, i.e., has not been logically deleted.

ename	eno	street	city	birthday	
Franziska	6542	Rennweg 683	Zurich	1963-07-04	...
Franziska	6542	Rennweg 683	Zurich	1963-07-04	...
Lilian	3463	46 Speedway	Tucson	1970-03-09	...

	Valid	Transaction
...	[1995-02-01 - 1995-07-01)	[1995-07-01 - 9999-12-31)
...	[1996-01-01 - 9999-12-31)	[1995-07-01 - 9999-12-31)
...	[1995-02-02 - 9999-12-31)	[1995-07-01 - 9999-12-31)

We retain the **salary** table as a table with valid-time support.

Temporal upward compatibility guarantees that conventional, nontemporal queries, updates, integrity constraints, etc. work as before, with the same semantics. We can list those for which (currently, as best known) no one makes a higher salary in a different city.

```
SELECT ename
FROM employee AS e1, salary AS s1
WHERE e1.eno = s1.eno
   AND NOT EXISTS (SELECT ename
                   FROM employee AS e2, salary AS s2
                   WHERE e2.eno = s2.eno AND s2.amount > s1.amount
                      AND e1.city <> e2.city)
```

This takes a timeslice in both valid time and transaction time at now, and returns the result: Lilian.

We can also ask, for all time, when this is true, by simply prepending "VALIDTIME."

```
VALIDTIME SELECT ename
FROM employee AS e1, salary AS s1
WHERE e1.eno = s1.eno
   AND NOT EXISTS (SELECT ename
                   FROM employee AS e2, salary AS s2
                   WHERE e2.eno = s2.eno AND s2.amount > s1.amount
                      AND e1.city <> e2.city)
```

This returns a table with valid-time support, evaluated with sequenced valid semantics, after the current transaction timeslice has been taken.

ename	Valid
Franziska	[1995-02-01 - 1995-02-02)
Lilian	[1995-02-02 - 1995-04-01)
Lilian	[1995-04-01 - 9999-12-31)

There are two rows for Lilian, because two rows of `salary` participated in computing the result. Interestingly, Franziska satisfied the where condition for exactly one day, before Lilian was hired.

Temporally upward compatible modifications also work as before. Assume it is now August 1, 1995. Franziska just moved.

```
UPDATE employee
SET street = 'Niederdorfstrasse 2'
WHERE ename = 'Franziska'
COMMIT
```

This update yields the following `employee` table. Note that although Franziska is at the new address starting on August 1, 1995, since she wo not be an employee for the next five months, her new address is recorded from January 1, 1996 onward.

ename	eno	street	city	birthday		Valid	Transaction
Franziska	6542	Rennweg 683	Zurich	1963-07-04	...	[1995-02-01 - 1995-07-01)	[1995-07-01 - 9999-12-31)
Franziska	6542	Rennweg 683	Zurich	1963-07-04	...	[1996-01-01 - 9999-12-31)	[1995-07-01 - 1995-08-01)
Franziska	6542	Niederdorfstrasse 2	Zurich	1963-07-04	...	[1996-01-01 - 9999-12-31)	[1995-08-01 - 9999-12-31)
Lilian	3463	46 Speedway	Tucson	1970-03-09	...	[1995-02-02 - 9999-12-31)	[1995-07-01 - 9999-12-31)

Since the history of the database is recorded in tables with both valid-time and transaction-time support, we can find out when corrections were made, using a nonsequenced transaction query. Assume it is now September 1, 1995.

The query "When was the street corrected, and what were the old and new values?" combines nonsequenced transaction semantics with sequenced valid semantics.

```
NONSEQUENCED TRANSACTIONTIME AND VALIDTIME
SELECT e1.ename, e1.street AS old_street, e2.street AS new_street,
       BEGIN(TRANSACTIONTIME(e2)) AS trans_time
FROM employee AS e1, employee AS e2
WHERE e1.eno = e2.eno
   AND TRANSACTIONTIME(e1) MEETS TRANSACTIONTIME(e2)
```

This yields the following table with valid-time support. The `trans_time` column specifies when the change was made; the implicit timestamp indicates the valid-time period of the fact that was changed.

ename	old_street	new_street		trans_time	Valid
Franziska	Rennweg 683	Niederdorfstrasse 2	...	1995-08-01	[1996-01-01 - 9999-12-31)

To extract all the information from the employee table, we can use a sequenced valid/sequenced transaction query.

```
VALIDTIME AND TRANSACTIONTIME SELECT * FROM employee
```

Modifications take effect at the current transaction time. However, we can still specify the scope of the change in valid time, both before and after now (retroactive and postactive changes, respectively).

Assume it is now October 1, 1995. Lilian moved last June 1.

```
VALIDTIME PERIOD '[1995-06-01 - 9999-12-31)' UPDATE employee
SET street = '124 Alberca'
WHERE ename = 'Lilian'
COMMIT
```

This update yields the following employee table.

ename	eno	street	city	birthday	
Franziska	6542	Rennweg 683	Zurich	1963-07-04	...
Franziska	6542	Rennweg 683	Zurich	1963-07-04	...
Franziska	6542	Niederdorfstrasse 2	Zurich	1963-07-04	...
Lilian	3463	46 Speedway	Tucson	1970-03-09	...
Lilian	3463	46 Speedway	Tucson	1970-03-09	...
Lilian	3463	124 Alberca	Tucson	1970-03-09	...

	Valid	Transaction
...	[1995-02-01 - 1995-07-01)	[1995-07-01 - 9999-12-31)
...	[1996-01-01 - 9999-12-31)	[1995-07-01 - 1995-08-01)
...	[1996-01-01 - 9999-12-31)	[1995-08-01 - 9999-12-31)
...	[1995-02-02 - 9999-12-31)	[1995-07-01 - 1995-10-01)
	[1995-02-02 - 1995-06-01)	[1995-10-01 - 9999-12-31)
	[1995-06-01 - 9999-12-31)	[1995-10-01 - 9999-12-31)

Finally, arbitrarily complex queries in transaction time can be expressed with nonsequenced transaction queries.

The query, "When was an employee's address for 1995 corrected?", involves nonsequenced transaction semantics and sequenced valid semantics, with a temporal scope of 1995. Assume that it is November 1, 1995.

```
NONSEQUENCED TRANSACTIONTIME AND VALIDTIME
             PERIOD '[1995-01-01 - 1996-01-01)'
SELECT e1.ename, e1.street AS old_street, e2.street AS new_street,
       BEGIN(TRANSACTIONTIME(e2)) AS trans_time
FROM employee AS e1, employee AS e2
WHERE e1.eno = e2.eno
      AND TRANSACTIONTIME(e1) MEETS TRANSACTIONTIME(e2)
      AND e1.street <> e2.street
```

This evaluates to the following result, which has an explicit column denoting the date the change was made, and an implicit valid time indicating the time in reality in question.

ename	old_street	new_street	trans_time	Valid
Lilian	46 Speedway	124 Alberca	1995-10-01	[1995-06-01 - 1996-01-01)

Note that the period from February through May is not included in the valid time, as the street did not change for that period.

As always, the concepts also apply to views, cursors, constraints, and assertions.

In Sec. 5.3 we gave an example of a sequenced constraint (`VALIDTIME CHECK (amount > 1000 AND amount < 12000)`) on the `salary` table. This constraint must hold independently on every (valid-time) state of the table. In Sec. 5.4 we gave a series of valid-time constraints on the `ename` column of the `employee` table. Those alternatives apply orthogonally to the transaction time. As an example, the assertion, "An entry in the security table can never be updated. It can only be deleted, and a new entry, with another key value, inserted.", can be expressed with a nonsequenced transaction semantics, stating in effect that the key value is unique over all transaction time.

```
CREATE TABLE security (
        keyvalue NUMERIC(8) NONSEQUENCED TRANSACTIONTIME UNIQUE,
        ...
)
```

7 Comparison with the UK Proposal

We end by comparing the above constructs, termed the US proposal, with the UK proposal [18], which has been incorporated into Part 7, SQL/Temporal [8], by applying them to the simple case study introduced in Secs. 3 and 6. This comparison will revisit and exemplify many of the salient points made earlier. These examples illustrate that SQL/Temporal could be extended in a minimal fashion along the lines discussed in this paper to provide much better support for temporal applications.

1. An `employee` table has five columns, `ename`, `eno`, `street`, `city`, and `birthdate`. The related salary table has two columns, `eno` and `amount`. Column `salary.eno` is a foreign key referencing the column `employee.eno`.

   SQL without time:

   ```
   CREATE TABLE employee(ename VARCHAR(12),
             eno INTEGER PRIMARY employee,
             street VARCHAR(22), city VARCHAR(10), birthday DATE)

   CREATE TABLE salary(eno INTEGER PRIMARY KEY REFERENCES
             employee, amount INTEGER)
   ```

US proposal with time: (discussed in this paper):

```
CREATE TABLE employee(ename VARCHAR(12),
          eno INTEGER VALIDTIME PRIMARY KEY,
          street VARCHAR(22), city VARCHAR(10), birthday DATE)
AS VALIDTIME PERIOD(DATE)

CREATE TABLE salary(eno INTEGER VALIDTIME PRIMARY KEY
                    VALIDTIME REFERENCES employee,
          amount INTEGER)
AS VALIDTIME PERIOD(DATE)
```

"AS VALIDTIME PERIOD(DATE)" specifies that an unnamed column, maintained by the DBMS, will contain the row's timestamp. "VALIDTIME" specifies that the integrity constraints (primary key, referential integrity) are to apply at each instant (in this case, each day).

UK proposal with time:

```
CREATE TABLE employee(ename VARCHAR(12), eno INTEGER,
          street VARCHAR(22), city VARCHAR(10), birthday DATE,
          When PERIOD(DATE))

CREATE TABLE salary(eno INTEGER,
          amount INTEGER, When PERIOD(DATE))
```

The UK proposal does not have support for referential integrity for such tables, nor for primary key constraints (adding `When` to the primary key does not work). Additional syntax is needed. Currently the only way to do this is with complex `ASSERTION`s, left as an exercise for the reader.

2. "List the history of those employees who have *or had* no salary."

SQL without time:

```
SELECT ename
FROM employee
WHERE eno NOT IN (SELECT eno FROM salary)
```

US proposal:

```
VALIDTIME SELECT ename
FROM employee
WHERE eno NOT IN (SELECT eno FROM salary)
```

To get the history of *any* query using the US proposal, simply prepend `VALIDTIME`. The change proposal and public-domain prototype demonstrate that the semantics may be implemented via a period-based algebra. The large body of performance-related research in temporal databases is applicable to implementing this semantics.

UK proposal:

```
WITH E1 AS (SELECT eno, ename, EXPAND(When) AS EW FROM employee)
WITH S1 AS (SELECT eno, EXPAND(When) AS EW FROM salary)
```

```
SELECT ename, PERIOD [ When, When ] AS When
FROM E1, TABLE(E1.EW) AS E2(When)
WHERE eno NOT IN (SELECT S1.eno
                  FROM S1, TABLE(S1.EW) AS S2(When)
                  WHERE S2.When = E2.When)
NORMALIZE ON When
```

The semantics of EXPAND is to duplicate each row of the argument table for each granule (day) in the When period. Once this table has been expanded, perform the NOT IN individually, for each day (examining only those salary rows valid on the day in question), then NORMALIZE the When column back to a period (collecting contiguous days into a single period).
If each row is valid on average for one year, then the result of the equijoin of E1 and E2 will have 360 *times* the number of rows of employee, with a dramatic decrease in performance. Changing the granularity to second generates additional tuples on the order of a factor 10^5, which could seriously affect performance. The approach of using EXPANDING does not work here, because the aggregate should be evaluated between the EXPAND and the NORMALIZE. The UK committee has provided a construct, EXCEPT EXPANDING, which can also be used to express this particular special case. The user can take the original SQL query, above, and map it into the relational algebra, with NOT IN being mapped to relation difference.

$$\pi_{\texttt{ename}}(\texttt{employee} \bowtie (\pi_{\texttt{eno}}(\texttt{employee}) - \pi_{\texttt{eno}}(\texttt{salary})))$$

Then the user can map this back into SQL.

```
SELECT ename
FROM employee
WHERE eno IN (SELECT eno FROM employee
              EXCEPT SELECT eno FROM salary)
```

As a third step, the user can map this into a temporal query using EXPAND and NORMALIZE.

```
WITH E1 AS (SELECT ename, eno, EXPAND(When) AS EW FROM employee),
     E2 AS (SELECT eno, EXPAND(When) AS EW
            FROM (SELECT eno FROM employee
                  EXCEPT EXPANDING(When)
                  SELECT eno FROM salary) AS E3)
SELECT ename, PERIOD [ E3.When, E3.When ] AS When
FROM E1, TABLE(E1.EW) AS E3(When)
WHERE E1.eno IN (SELECT eno
                 FROM E2, TABLE(E2.EW) AS E4(When)
                 WHERE E1.eno = E2.eno AND E3.When = E4.When)
NORMALIZE ON When
```

This trick of using EXCEPT can also be applied with the US proposal, but omitting the complex third step and the EXPANDs and NORMALIZEs entirely.

```
VALIDTIME SELECT ename
FROM employee
WHERE eno IN (SELECT eno FROM employee
              EXCEPT SELECT eno FROM salary)
```

All of the UK alternatives have the problem (not shared by the US alternatives) that if the left-hand table has duplicates, then NORMALIZE will automatically remove them, yielding an incorrect result (the original SQL query did not specify DISTINCT). It is an exercise to the reader to show how this English query can be correctly expressed using an explicit When column. It *is* possible to do so, but it is exceedingly difficult.

There have been essentially no results published on how to optimize queries with expansion or normalize operations. Also, no general procedure has been provided for converting an arbitrary, non-temporal query into its temporal analogue using the UK constructs. Finally, while EXCEPT EXPANDING has been provided, nó other expanding variants have been defined for other relational operators. In contrast, in the US proposal a sequenced variant of *any* query can be specified by prepending the VALIDTIME reserved word.

3. "Give the history of the number of highly-paid employees in each city."

SQL without time:

```
SELECT city, COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
GROUP BY city
```

US proposal:

```
VALIDTIME SELECT city, COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
GROUP BY city
```

The VALIDTIME specifies that we are interested in the time-varying count. The syntax is declarative. The semantics is specified on a row-by-row basis; changing the granularity from day to second will not impact its performance.

UK proposal:

```
WITH E1 AS (SELECT eno, city, EXPAND(When) AS EW FROM employee),
        S1 AS (SELECT eno, EXPAND(When) AS EW
               FROM salary
               WHERE amount > 5000)
```

```
SELECT city, COUNT(*), PERIOD [ E2.When, E2.When ] AS When
FROM E1, TABLE(E1.EW) AS E2(When), S1, TABLE(S1.EW) AS S2(When)
WHERE E2.When = S2.When AND E1.eno = S1.eno
GROUP BY city, When
NORMALIZE ON When
```

The syntax is procedural: first expand, then execute the select, then normalize. The EXPAND operator generates a SET of DAYs, which is then used to duplicate the rows of employee, one for each day each row is valid (the join in the from clause). The GROUP BY ensures that the COUNT is performed separately for each day. The NORMALIZE converts the many rows, one for each day, into periods.

4. "Give Therese a salary of $6,000 for 1994."

   SQL without time:

```
UPDATE salary
SET amount = 6000
WHERE eno IN (SELECT eno FROM employee WHERE ename = 'Therese')
```

   US proposal:

```
VALIDTIME PERIOD '[1994-01-01 - 1994-12-31]' UPDATE salary
SET amount = 6000
WHERE eno IN (SELECT eno FROM employee WHERE ename = 'Therese')
```

   UK proposal:
   The UK proposal has no support for this operation. Instead, each row must be examined to determine the overlap with 1994, and adjusted with an UPDATE and two INSERT statements. This is left as an exercise for the reader.

5. To know when rows are inserted and (logically) deleted, we add transaction-time support.

   US proposal:

```
ALTER TABLE employee ADD TRANSACTIONTIME
ALTER TABLE salary ADD TRANSACTIONTIME
```

   Since transaction time is automatically managed by the DBMS, system integrity is ensured. Due to temporal upward compatibility, the integrity constraints work as before, as do updates, such as the one above.

   UK proposal:

```
ALTER TABLE employee ADD COLUMN InsertTime TIMESTAMP(3)
DEFAULT CURRENT_TIMESTAMP

ALTER TABLE employee ADD COLUMN DeleteTime TIMESTAMP(3)
DEFAULT NULL
```

```
ALTER TABLE salary ADD COLUMN InsertTime TIMESTAMP(3)
DEFAULT CURRENT_TIMESTAMP

ALTER TABLE salary ADD COLUMN DeleteTime TIMESTAMP(3)
DEFAULT NULL
```

There is no support for transaction time in the UK proposal. There is no way to ensure that the application correctly manages the information in these two columns. System integrity can easily be compromised. Adding these two columns also breaks the primary key and referential integrity constraints. Such constraints must be reformulated as complex assertions that take the three time columns into account.
Updates are more complicated when these additional columns are present.

6. "How many highly-paid employees are in each city?"

SQL without time:

```
SELECT city, COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
GROUP BY city
```

US proposal:

```
SELECT city, COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
GROUP BY city
```

This still works, because the default is to take the currently valid data that has not been deleted or updated (temporally upward compatible in both valid and transaction time).

UK proposal:

```
WITH E1 AS (SELECT eno, city FROM employee
      WHERE DeleteTime IS NULL AND CURRENT_DATE OVERLAPS When),
     S1 AS (SELECT eno FROM salary
      WHERE DeleteTime IS NULL AND CURRENT_DATE OVERLAPS When
              AND amount > 5000)
SELECT city, COUNT(*)
FROM E1, S1
WHERE E1.eno = S1.eno
GROUP BY city
```

Since temporal upward compatibility is not satisfied by the UK proposal, the user must explicitly select the current information.

To get the *history* of the number of highly-paid employees in each city, some changes are required.

US proposal:

```
VALIDTIME SELECT city, COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
GROUP BY city
```

We retain temporal upward compatibility in transaction time (i.e., the data that has not been deleted or updated), but specify sequenced valid semantics to get the history, via VALIDTIME.

UK proposal:

```
WITH E1 AS (SELECT eno, city, EXPAND(When) AS EW
            FROM employee
            WHERE DeleteTime IS NULL),
     S1 AS (SELECT eno, EXPAND(When) AS EW
            FROM salary
            WHERE DeleteTime IS NULL AND amount > 5000)
SELECT city, COUNT(*), PERIOD [ E2.When, E2.When ] AS When
FROM E1, TABLE(E1.EW) AS E2(When), S1, TABLE(S1.EW) AS S2(When),
WHERE E1.eno = S1.eno AND E2.When = S2.When
GROUP BY city, When
NORMALIZE ON When
```

The user must explicitly select the currently stored information in transaction time ("WHERE DeleteTime IS NULL") and must EXPAND and NORMALIZE to compute the aggregate.

7. "When did we think that there were many (> 25) highly-paid employees in Tucson?"

US proposal:

```
TRANSACTIONTIME SELECT COUNT(*)
FROM employee, salary
WHERE employee.eno = salary.eno AND amount > 5000
        AND city = 'Tucson'
GROUP BY city
HAVING COUNT(*) > 25
```

TRANSACTIONTIME specifies that we wish to look over past states of the table. VALIDTIME is not specified, as we want to know only about the information about current employees. The execution is on a row-by-row basis, and is independent of both the valid time and transaction time granularities.

UK proposal:

```
WITH E1 AS (SELECT eno, EXPAND(WhenP) AS EW
            FROM (SELECT eno, PERIOD(InsertTime, DeleteTime)
                         AS WhenP
                  FROM employee
                  WHERE CURRENT_TIMESTAMP OVERLAPS When
                     AND city = 'Tucson') AS ET),
     S1 AS (SELECT eno, EXPAND(WhenP) AS EW
            FROM (SELECT eno, PERIOD(InsertTime, DeleteTime)
                         AS WhenP
                  FROM salary
                  WHERE CURRENT_DATE OVERLAPS When
                     AND amount > 5000) AS ET)
SELECT COUNT(*), PERIOD [ E2.When, E2.When ] AS When
FROM E1, TABLE(E1.EW) AS E2(When), S1, TABLE(S1.EW) AS S2(When)
WHERE E1.eno = S1.eno AND E2.When = S2.When
GROUP BY When
HAVING COUNT(*) > 25
NORMALIZE ON When
```

The transaction time granularity is generally no coarser than a millisecond. Compared with the US proposal, this query will expand into $3 \cdot 10^{10}$ times the number of rows in the `employee` table. The `salary` table will be similarly exploded, then a join on the two tables taken. It is not clear how to optimize this query, as the result could change at any millisecond: the aggregate must be computed for each millisecond. It is doubtful that the UK query can even be computed with currently known query optimization/evaluation technology.

8 Summary

In this paper, we first outlined several desirable features of SQL/Temporal relative to SQL3: upward compatibility, temporal upward compatibility, and sequenced semantics. A series of four levels of increasing functionality was elaborated. The specific syntactic additions were outlined and examples given to illustrate these constructs. The extensions involve (a) the use of the `VALIDTIME` and `TRANSACTIONTIME` reserved words, to indicate valid-time, resp. transaction-time, support (in the case of schema specification statements) and sequenced semantics (in the case of queries, modifications, views, cursors, assertions and constraints), (b) the use of the `NONSEQUENCED` reserved word for nonsequenced semantics, and (c) the use of a period expression to temporally scope sequenced and nonsequenced queries, modifications, views, cursors, constraints, and assertions. In the change proposals now before the SQL3 committees [14, 15], we provide a formal semantics, in terms of the formal semantics of SQL3, that satisfied the sequenced semantics correspondence between temporal queries and

snapshot queries, and also provide the semantics for nonsequenced queries. In those change proposals we also list alternative implementation approaches which vary in the degree of implementation difficulty and the achievable performance. The implementation alternatives all compute the result by manipulating periods, and thus their performance is independent of the granularity of the underlying tables.

We also introduced tables with transaction-time support, sequenced transaction semantics, nonsequenced transaction semantics, scoping on transaction time via an optional period expression, and modification semantics. The specific syntactic additions were outlined and examples given to illustrate these constructs.

We end by listing some of the advantages of the approach espoused here.

- Upward compatibility is assured, permitting existing constructs to operate exactly as before.
- Only three new reserved words, `NONSEQUENCED`, `VALIDTIME`, and `TRANSACTIONTIME`, are required.
- Satisfaction of temporal upward compatibility ensures that existing applications do not break when tables without temporal support have such support added.
- The availability of sequenced semantics ensures that temporal queries, modifications, views, assertions, and constraints are easy to formalize, write and implement.
- Nonsequenced semantics permits tables with temporal support to be converted to tables without such support, with explicit timestamp columns, and for temporal support to be added to tables, even within a query.
- A simple period expression permits the temporal scope to be specified.
- The transaction-time extensions are compatible with, and orthogonal to, those for valid time.
- A public-domain prototype [16] demonstrates the practical viability of the proposed constructs. The quick tour was validated on this prototype.

We note that none of these benefits accrue from the UK proposal.

Acknowledgments

The inspiration for the constructs described here and proposed for incorporation into SQL/Temporal is the TSQL2 language. The participation of the TSQL2 Language Design Committee, which included Ilsoo Ahn, Gad Ariav, Don S. Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T.Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo and Surynarayana M. Sripada, was critical.

David Toman provided helpful comments on a previous draft. We also appreciate the extensive feedback from the ANSI and ISO SQL3 committees, which helped shape the specifics of this proposal.

This research was supported in part by the National Science Foundation through grants ISI-9202244 and IRI-9632569, by grants from IBM, the AT&T Foundation, and DuPont, by the Danish Technical and Natural Science Research Councils through grants 9700780 and 9400911, respectively, and by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, as a Networks Activity of the Training and Mobility of Researchers Programme, contract no. FMRX-CT96-0056.

References

1. Bair, J., M. Böhlen, C.S. Jensen, and R.T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages," *Business Informatics* (in German, *Wirtschaftsinformatik*) 39(1):25–34, February 1997.
2. Böhlen, M. H., C. S. Jensen and R. T. Snodgrass,. "Evaluating the Completeness of TSQL2," in *Proceedings of the VLDB International Workshop on Temporal Databases.* Ed. J. Clifford and A. Tuzhilin. Springer Verlag, September 1995, pp. 153–172.
3. Böhlen, M. H. and C. S. Jensen. *Seamless Integration of Time into SQL.* Technical Report R-962049, Aalborg University, Department of Computer Science, Denmark, December 1996.
4. Gadia, S. K. "A Homogeneous Relational Model and Query Languages for Temporal Databases." *ACM Transactions on Database Systems* 13(4):418–448, December 1988.
5. Jackson, M. A. *System Development.* Prentice-Hall International Series in Computer Science. Prentice-Hall International, Inc., 1983.
6. Jensen, C. S. and R. Snodgrass, "Temporal Specialization and Generalization." *IEEE Transactions on Knowledge and Data Engineering* 6(6):954–974, December 1994.
7. Jensen, C. S., J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes and S. Jajodia (eds). "A Glossary of Temporal Database Concepts." *ACM SIGMOD Record* 23(1):52–64, March 1994.
8. Melton, J. (ed.) *SQL/Temporal.* July, 1997. (ISO/IEC JTC 1/SC 21/WG 3 DBL-LGW-013.)
9. Pissinou, N., R. T. Snodgrass, R. Elmasri, I. S. Mumick, M. T. Özsu, B. Pernici, A. Segev, and B. Theodoulidis, "Towards an Infrastructure for Temporal Databases: Report of an Invitational ARPA/NSF Workshop," *SIGMOD Record* 23(1):35–51, March, 1994.
10. Snodgrass, R.T., I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo, and S.M. Sripada. "TSQL2 Language Specification," *ACM SIGMOD Record* 23(1):65–86, March, 1994.
11. Snodgrass, R. T. and H. Kucera. *Rationale for Temporal Support in SQL3.* 1994. (ISO/IEC JTC1/SC21/WG3 DBL SOU-177, SQL/MM SOU-02.)
12. Snodgrass, R. T., K. Kulkarni, H. Kucera and N. Mattos. *Proposal for a new SQL Part—Temporal.* 1994. (ISO/IEC JTC1/SC21/WG3 DBL RIO-75, X3H2-94-481.)
13. Snodgrass, R. T. (editor), Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo and Suryanarayana M. Sripada. *The Temporal Query Language TSQL2.* Kluwer Academic Pub., 1995.

14. Snodgrass, R. T., M. H. Böhlen, C. S. Jensen and A. Steiner. *Adding Valid Time to SQL/Temporal*, change proposal, ANSI X3H2-96-501r2, ISO/IEC JTC 1/SC 21/WG 3 DBL-MAD-146r2, November 1996, 77 pages. At URL: <ftp://ftp.cs.arizona.edu/tsql/tsql2/sql3/mad146.ps> (version current November 21, 1996).
15. Snodgrass, R. T., M. H. Böhlen, C. S. Jensen and A. Steiner. *Adding Transaction Time to SQL/Temporal*, change proposal, ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-147r2, November 1996, 47 pages. At URL: <ftp://ftp.cs.arizona.edu/tsql/tsql2/sql3/mad147.ps> (version current November 21, 1996).
16. Steiner, A. and M. H. Böhlen. The TimeDB Temporal Database Prototype, Version 1.07, November 1996. At URL: <http://www.iesd.auc.dk/general/DBS/tdb/TimeCenter> or at URL: <ftp://ftp.cs.arizona.edu/tsql/timecenter/TimeDB.tar.gz> (version current March 26, 1997).
17. Tsotras, V. J. and A. Kumar. "Temporal Database Bibliography Update," *ACM SIGMOD Record* 25(1):41–51, March, 1996.
18. UK SQL Committee, *Expanded Table Operations.* 1996. (ISO/IEC JTC1/SC21/WG3 DBL MCI-67)
19. Yourdon, E. *Managing the System Life Cycle.* Yourdon Press, 1982.