ANSI X3H2-96-151r1
ISO/IEC JTC1/SC21/WG3 DBL MCI-142

I S O
INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

May 10, 1996

Subject: SQL/Temporal

Status: Change Proposal

Title: Adding Valid Time to SQL/Temporal

Source: ANSI Expert's Contribution

Authors: Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen and Andreas Steiner

Abstract: This change proposal specifies the addition of tables with valid-time support into SQL/Temporal, and explains how to use these facilities to migrate smoothly from a conventional relational system to a temporal system. 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. The proposal formally defines the semantics of the query language by providing a denotational semantics mapping to well-defined algebraic expressions. Several alternatives for implementing the language constructs are listed. A prototype system implementing these constructs on top of a conventional DBMS is publicly available.

References

[1] Böhlen, M. H. and Marti R. *On the Completeness of Temporal Database Query Languages*, in *Proceedings of the First International Conference on Temporal Logic*. D. M. Gabbay and H. J. Ohlbach, eds. Lecture Notes in Artificial Intelligence 827. Springer-Verlag, July 1994, pp. 283-300.

[2] Böhlen, M. H. *Valid-Time Integrity Constraints*, Aalborg University, October, 1995, 21 pages.

[3] 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. VLDB. Springer Verlag, Sep. 1995.

[4] Clifford, J., A. Croker and A. Tuzhilin. *On Completeness of Historical Relational Query Languages. ACM Transactions on Database Systems*, 19, No. 1, Mar. 1994, pp. 64–116.

[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, No. 6 (1994), pp. 954–974.

[7] Melton, J. (ed.) *SQL/Foundation*. March, 1996. (ISO/IEC JTC 1/SC 21/WG 3 DBL-MCI-002.)

[8] Melton, J. (ed.) *SQL/Temporal*. March, 1996. (ISO/IEC JTC 1/SC 21/WG 3 DBL-MCI-009.)

[9] Schueler, B. *Update Reconsidered*, in *Architecture and Models in Data Base Management Systems*. Ed. G. M. Nijssen. North Holland Publishing Co., 1977.

[10] Snodgrass, R. T., S. Gomez and E. McKenzie. *Aggregates in the Temporal Query Language TQuel. IEEE Transactions on Knowledge and Data Engineering*, 5, Oct. 1993, pp. 826–842.

[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) *The Temporal Query Language TSQL2*. Kluwer Academic Pub., 1995.

[14] Steiner, A. and M. H. Böhlen. The TimeDB Temporal Database Prototype, September, 1995. Available at `http://www.iesd.auc.dk/general/DBS/tdb/TimeCenter` or at `ftp://ftp.cs.arizona.edu/tsql/timecenter/TimeDB.tar.gz`.

[15] Tsichritzis, D.C. and F.H. Lochovsky. *Data Models*. Software Series. Prentice-Hall, 1982.

[16] Yourdon, E. *Managing the System Life Cycle*. Yourdon Press, 1982.

1 Introduction

This change proposal introduces additions to SQL/Temporal to add valid-time support to SQL3. We outline a four-level approach for the integration of time. We motivate and discuss each level in turn, and we define the syntactic extensions that correspond to each level. We will see that the 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 [14].

The proposed language constructs ensure temporal upward compatibility, sequenced valid semantics, and non-sequenced semantics, important properties that will be discussed in detail in Section 5.

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 three columns: Name, Manager and Dept. We then store historical information by adding a fourth column, When, of data type `PERIOD`. Manager is a foreign key for Employee.Name. This means that at each point in time, the character string value in the Manager column also occurs in the Name column (probably in a different row) at the same time. This cannot be expressed via SQL's foreign key constraint, which doesn't take time into account. Formulate this constraint instead as an assertion.

- Consider the query "List those employees who are not managers." This can easily be expressed in SQL, using `EXCEPT` or `NOT EXISTS`, on the original, three-column table. Things are just a little harder with the When column; a where predicate is required to extract the current employees. Now formulate the query "List those employees who were not managers, and indicate when." `EXCEPT` and `NOT EXISTS` won't work, because they don't consider time. This simple temporal query is challenging even to SQL experts.

- Consider the query "Give the number of employees in each department." Again, this is a simple query in SQL. Formulate the query "Give *the history of* the number of employees in each department." This query is extremely difficult without temporal support in the language.

- Now formulate the modification "Change the manager of the tools department for 1994 to Bob." 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 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 in this change proposal, the above queries can all be easily written by an intermediate SQL programmer in about five minutes.

Referential integrity can be specified using sequenced valid semantics (which will be defined, exemplified, and provided a formal definition later in this document):

```
CREATE TABLE Employee(
   Name    VARCHAR(30),
   Manager VARCHAR(30) VALIDTIME REFERENCES Employee (Name),
   Dept    VARCHAR(20)) AS VALIDTIME DAY
```

Here we indicate that the table has valid-time support through "`AS VALIDTIME DAY`" and that the referential integrity is to hold for each point in time through "`VALIDTIME REFERENCES`".

For the query "List those employees who are not managers," 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 Name FROM Employee EXCEPT SELECT Manager FROM Employee
```

This results in a conventional table, with one column.

We use sequenced valid semantics in the query "List those employees who were not managers, and when."

```
VALIDTIME SELECT Name FROM Employee EXCEPT SELECT Manager FROM Employee
```

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 be a manager, whereas at other times, the employee is a manager. A one-column table results, but this time with valid-time support (i.e., the periods of time when each was not a manager is included).

The query "Give the number of employees in each department" is easy given temporal upward compatibility.

```
SELECT Dept, COUNT(*)
FROM Employee
GROUP BY Dept
```

Again, we just get the current count for each department. To extract "*the history of* the number of employees in each department", only a simple change is required.

```
VALIDTIME SELECT Dept, COUNT(*)
FROM Employee
GROUP BY Dept
```

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

Modifications work in similar ways. The modification "Change the manager of the tools department for 1994 to Bob" can be expressed by following `VALIDTIME` with a period expression.

```
VALIDTIME PERIOD '[1994-01-01 - 1994-12-31]' UPDATE Employee
SET Manager = 'Bob'
WHERE Dept = 'Tools'
```

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.

4 Scope

Research on temporal databases has identified several properties crucial to temporal database systems, including support for valid-time, transaction time, temporal aggregates, indeterminacy, time granularity, user-defined calendars, vacuuming, and schema versioning. This document is the second in a series that will propose constructs for SQL/Temporal drawn from the consensus temporal query language TSQL2 [13]. The first [12], which was accepted in July, 1995, concerned the `PERIOD` data type.

The present change proposal addresses support for valid-time, specifically temporal upward compatibility, sequenced valid, and nonsequenced valid support. The next proposal will add support for transaction time. Future proposals will concern time granularities, temporal indeterminacy, and other features relevant to SQL3 that are fully supported in TSQL2. However, it is important that each proposal be comprehensive in its motivation of the additions, its presentation of the syntactic changes, and its specification of the semantics of the new constructs. For this reason, each change proposal should be separately considered and evaluated by the SQL3 standards committees.

While the language additions proposed here are modest, the productivity gains made available to the application programmer are significant. In particular, we will show how adding a single reserved word will convert any conventional (termed *snapshot*) query into a temporal query that extracts the history of the aspect being queried. This permits users to express rather complex temporal queries easily, by first formulating them as snapshot queries, then adding the reserved word. This parallel will be exploited in the semantics, permitting *any* SQL3 query to be rendered temporal. Moreover the syntactic modification not only holds for queries but also for view definitions, insert statements, delete statements, update statements, cursor declarations, table constraint definitions, column constraint definitions, and the definition of assertions.

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.

5 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 to the new DBMS that must be satisfied in order to avoid different potential problems when migrating. Formal definitions of these requirements may be found in Appendix A.

5.1 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 Figure 1, a conventional table is

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

denoted with a rectangle. 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.

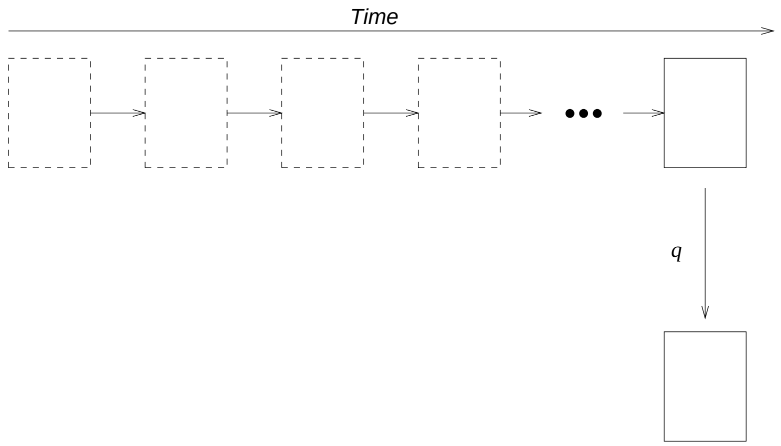

Figure 1: Level 1 evaluates an SQL3 query over a table without temporal support and returns a table also without temporal support

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* constructs and semantics), it is relatively easy to ensure that SQL/Temporal is upward compatible with SQL3.

Throughout, we provide examples of the various levels. In Section 6, 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. □

5.2 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 doesn't 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 Figure 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.

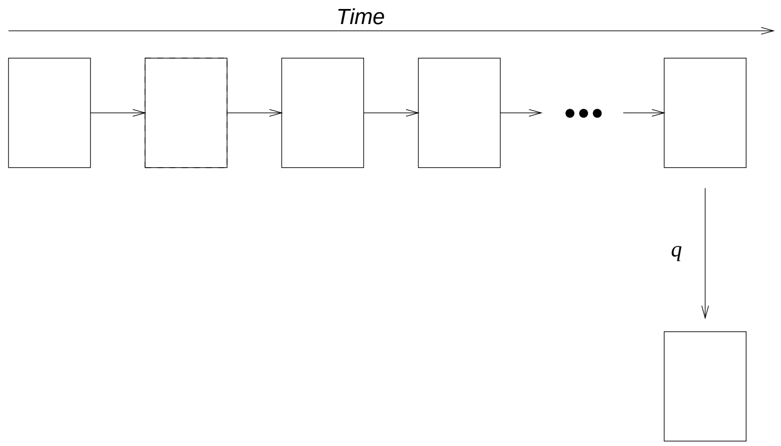

Figure 2: Level 2 evaluates an SQL3 query over a table with valid-time support and returns a table with similar support

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 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 [16]); 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 data with temporal. 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.

5.3 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.

Figure 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.

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

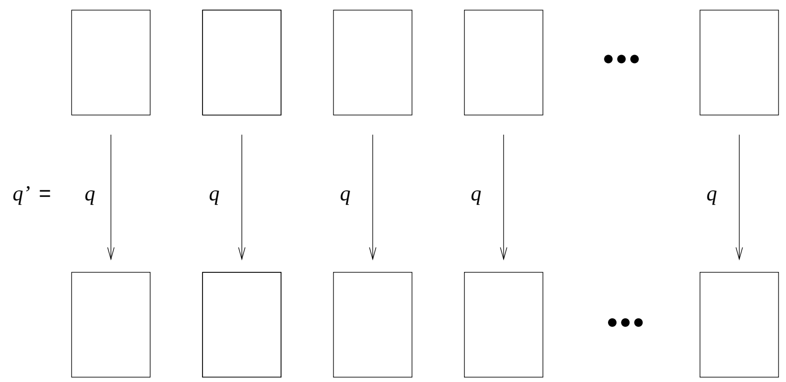

Figure 3: Level 3 evaluates an SQL/Temporal query over a table with valid-time support and returns a table with similar support

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 Figure 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 modification statement that it generalizes.

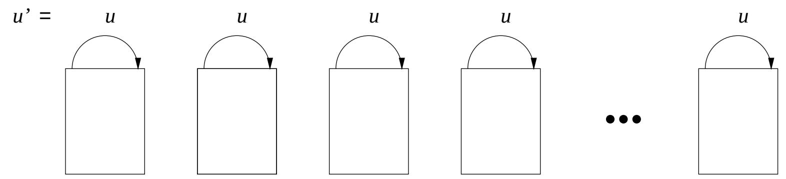

Figure 4: Level 3 also evaluates an SQL/Temporal modification on a table with valid-time support

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. □

5.4 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 Figure 5, in which each state of the resulting table requires information from possibly all states of the source table.

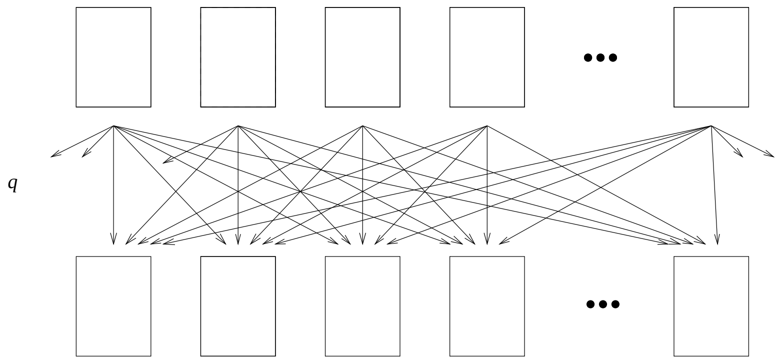

Figure 5: Level 4 evaluates a non-sequenced SQL/Temporal query over a table with valid-time support and returns a table with similar support

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 Figure 6, each state of the table with valid-time support is possibly modified, using information from possibly all states of the table before the modification. Non-sequenced modifications include future modifications.

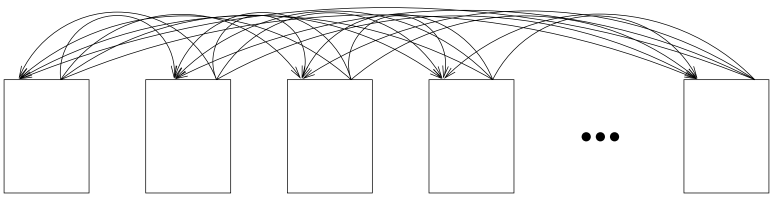

Figure 6: Level 4 also evaluates a non-sequenced SQL/Temporal modification on a table with valid-time support

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. □

5.5 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 temporal statements be a sequenced extension of the existing statements guarantees that the 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, as shown in Section 7, and enables a wide range of implementation alternatives, some of which are listed in Section 8.2.

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.

6 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 Section 5, to support easy migration of legacy SQL3 application code [3]. 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 [14]. The reader may find it instructive to execute the sample statements on the prototype. In the examples, executable statements are displayed in `typewriter style` on a line of their own starting with the prompt "`>` ".

6.1 Level 1: Upward Compatibility

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

6.1.1 SQL3 Extensions

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

6.1.2 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 REFERENCES employee(eno), 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;
```

6.2 Level 2: Temporal Upward Compatibility

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

6.2.1 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 DAY`". 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.

6.2.2 A Quick Tour

The following statements are executed on February 1, 1995.

```
> ALTER TABLE salary ADD VALIDTIME DAY;
> ALTER TABLE employee ADD VALIDTIME DAY;
```

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 *forever*, 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;
```

6.3 Level 3: Sequenced Language Constructs

Level 3 adds syntactically similar, sequenced counterparts of existing queries, modifications, views, constraints, and assertions (see Figure 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.

6.3.1 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; Section 10 supplies the details.

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 all states 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.

Fetch statement The period associated with a row with valid-time support can be placed in a local variable in embedded SQL.

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.

6.3.2 A Quick Tour

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 didn't join the company until February 2).

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 *proactive* 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 #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.

6.4 Level 4: Non-Sequenced Language Constructs

Level 4 accounts for non-sequenced queries (see Figure 5) and non-sequenced modifications (see Figure 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.

6.4.1 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; Section 10 supplies the details.

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 simultaneously to all 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 `NONSEQUENCED VALIDTIME`, such constraints must apply to the table with valid-time support as a whole.

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 simultaneously to all states comprising the table with valid-time support.

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

This portion includes other useful, related constructs.

- 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.
- 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` and `VALIDTIME NONSEQUENCED` specifies the temporal scope of the modification: the times at which the modification is to be applied.
- The value expression "`VALIDTIME(`<correlation name>`)`" 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.

6.4.2 A Quick Tour

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.

```
> 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);
```

This isn't quite correct, because the period expression following `VALIDTIME` can only mention the columns of the following select statement. 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;
```

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.

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 (at least, those 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 and S2.eno = S.eno) AS S3
                    );
> 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, if 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.

```
> 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.

7 Formal Semantics of SQL/Temporal

In this section, we provide a formal semantics for the constructs introduced into SQL/Temporal, expressed in terms of the relational algebraic semantics for SQL3.

We use $\langle t \| VT \rangle$ to denote a row in a table with valid-time support. The vertical double-bar "$\|$" is used to separate valid-time from explicit attributes. If VT is a period, then VT^- is its beginning delimiting timestamp and VT^+ is the granule following its ending delimiting timestamp.

7.1 Translating SQL/Temporal Queries to Relational Algebra Expressions

We first provide the semantics of an SQL3 query over tables without temporal support. In the definition given next, let $r_1, \ldots, r_n$ denote tables without temporal support. We base the definition of the semantics on the semantics of SQL3, expressed in terms of the relational algebra.

$$[\![\text{<query expression>}]\!]_{\text{SQL/T}}(r_1, \ldots, r_n) \triangleq [\![\text{<query expression>}]\!]_{\text{standard}}(r_1, \ldots, r_n)$$

Here, $[\![\text{<query expression>}]\!]_{\text{standard}}$, which evaluates to the relational algebra expression that corresponds to <query expression>, is assumed to be given. This definition satisfies upward compatibility.

Example 8: We start with a non-temporal query, i.e., a query evaluated with standard semantics. Assume `p` and `q` are both tables without temporal support. The query Q_1

```
SELECT p.X
FROM p, q
WHERE p.X = q.X
```

is equivalent to the relational algebra expression

$$[\![Q_1]\!]_{\text{SQL/T}}(p, q) = [\![Q_1]\!]_{\text{standard}}(p, q) = \pi_{p.X}(p \bowtie_{p.X=q.X} q) \ .$$

□

The semantics of an SQL3 query over a combination of snapshot and tables with valid-time support is very similar. For every table with valid-time support r_i appearing as an argument, replace it with $\tau^{vt}_{now}(r_i)$ on the right hand side. The valid-timeslice operator τ^{vt}_c extracts the current snapshot state from a table with valid-time support.

$$\tau_c^{vt}(r) \quad \triangleq \quad \{t \mid \exists VT(\langle t \| VT\rangle \in r \wedge VT^- \leq c \wedge c < VT^+)\}$$

Example 9: We now examine a non-temporal query over a combination of tables with and without temporal support, with standard semantics. Assume p is a table without temporal support and t is a table with valid-time support. The query Q_1

```
SELECT p.X
FROM p, t
WHERE p.X = t.X
```

is equivalent to the relational algebra expression

$$[\![Q_1]\!]_{\mathrm{SQL/T}}(p,t) = [\![Q_1]\!]_{\mathrm{standard}}(p, \tau^{vt}_{now}(t)) = \pi_{p.X}(p \bowtie_{p.X=t.X} (\tau^{vt}_{now}(t))) \enspace .$$

□

This definition satisfies temporal upward compatibility.

Next, we define the semantics of sequenced SQL/Temporal additions in terms of the snapshot semantics. This allows these extensions to be consistent with all snapshot constructs defined in SQL3.

$$[\![\texttt{VALIDTIME} \text{ <query expression>}]\!]_{\mathrm{SQL/T}}(r_1, \ldots, r_n) \triangleq [\![\text{<query expression>}]\!]_{\mathrm{temporal}}(r_1, \ldots, r_n)$$

In this definition, $[\![\text{<query expression>}]\!]_{\mathrm{temporal}}$ is equivalent to $[\![\text{<query expression>}]\!]_{\mathrm{standard}}$, except that every non-temporal relational algebra operator (e.g., $\bowtie, \sigma, \pi$) is replaced by a corresponding temporal relational algebra operator (e.g., $\bowtie^{vt}, \sigma^{vt}, \pi^{vt}$). We provide definitions of the temporal algebra in Section 7.3.

Example 10: An SQL/Temporal query $Q_2 =$ VALIDTIME Q_1 is evaluated with temporal semantics, due to its leading valid clause. Both p and q must be tables with valid-time support. Thus,

```
VALIDTIME
  SELECT p.X
  FROM p, q
  WHERE p.X = q.X
```

is equivalent to the temporal relational algebra expression

$$[\![Q_2]\!]_{\mathrm{SQL/T}}(p,q) = [\![\texttt{VALIDTIME}\ Q_1]\!]_{\mathrm{SQL/T}}(p,q) = [\![Q_1]\!]_{\mathrm{temporal}}(p,q) = \pi^{vt}_{p.X}(p \bowtie^{vt}_{p.X=q.X} q) \enspace .$$

Note that apart from the ^{vt}-superscripts, which are added to relational algebra operators, the translation between SQL queries and relational algebra expressions has not changed at all. □

The definitions above satisfy sequenced semantics if the temporal relational operators are sequenced with respect to their conventional relational counterparts. The next step is to define a temporal relational algebra with this property.

7.2 The Conventional Relational Algebra

As a precursor to defining the temporal relational algebra, we review Codd's relational algebra.

$$
\begin{array}{lcl}
\sigma_c(r) & \triangleq & \{t \mid t \in r \wedge c(t)\} \\
\pi_f(r) & \triangleq & \{t_1 \mid t_2 \in r \wedge t_1 = f(t_2)\} \\
r_1 \cup r_2 & \triangleq & \{t \mid t \in r_1 \vee t \in r_2\} \\
r_1 \bowtie_c r_2 & \triangleq & \{t_1 \circ t_2 \mid t_1 \in r_1 \wedge t_2 \in r_2 \wedge c(\langle t_1 \circ t_2 \rangle)\} \\
r_1 \setminus r_2 & \triangleq & \{t \mid t \in r_1 \wedge t \notin r_2\} \\
AG_{agg,f}(r) & \triangleq & \{t \circ a \mid t \in r \wedge a = agg(\{t_1 | t_1 \in r \wedge f(t_1) = f(t)\})\}
\end{array}
$$

In this formalism, c is a predicate, f is a list of attributes (for the aggregate operator, a list of the `GROUP BY` attributes), and agg is a function (e.g., sum_3) that when applied to a set of rows returns the single value of the aggregate (e.g., `SUM`) evaluated over the indicated attribute (e.g., the third attribute).

Observe that the algebra defined above is based on sets and thus does not permit duplicates. We have chosen to assume a set-based framework in the semantics given here because this yields a short definition where the general approach stands out more clearly. The complications that follow from giving up the set-based basis have been explored in the past and are omitted. We emphasize that the proposed additions to SQL/Temporal do not impact the data model of SQL3 and are not strictly set based.

7.3 The Temporal Relational Algebra

The next step is to define the temporal relational algebra operators. Informally, each definition respects sequenced semantics. In addition to that, the algebra features two properties which we would like to point out. First, the algebra preserves the periods entered into the database, i.e., it *matters* for the query results whether we store, e.g., one row with a valid-time period of $[10-20]$ or two (value-equivalent) rows with valid-time periods $[10-15]$ and $[16-20]$, respectively. Second, care was taken to only consider end points of valid-time periods of rows when implementing the operators—intermediate time points are never used. This allows for an efficient (essentially, granularity independent) implementation.

In Figure 7, the constructor *intersect* (over two periods) returns a period containing those chronons in both underlying periods, and the predicate *overlaps* (over two periods) returns true if the two periods overlap and false, otherwise. Both operations are easily expressed as operations on the beginning and ending delimiting timestamps of periods. The symbol "$\circ$" denotes concatenation. The definition of AG^{vt} is especially complex. It determines *constant periods*, during which no row starts or ends [10]. A constant period can go from the start of one row to the start of another, from the start of one row to the end of another, or from the end of one row to the end of another.

7.4 Nonsequenced Semantics

When `NONSEQUENCED VALIDTIME` is used, each table with valid-time support is converted to a table without temporal support via the SN function.

$$SN(r) \triangleq \{\langle t, VT \rangle \mid \langle t \mid\mid VT \rangle \in r\}$$

Then, the query is evaluated with the conventional semantics. Assume in the following that all the tables are tables with valid-time support.

$$[\![\texttt{NONSEQUENCED VALIDTIME} <\text{query expression}>]\!]_{\text{SQL/T}}(r_1, \ldots, r_n)$$
$$\triangleq [\![<\text{query expression}>]\!]_{\text{standard}}(SN(r_1), \ldots, SN(r_n))$$

EXAMPLE 11: Let Q be the following non-temporal query.

```
SELECT p.X
FROM p, q
WHERE p.X = q.X
```

$$
\begin{array}{lll}
\sigma_c^{vt}(r) & \triangleq & \{\langle t\|VT\rangle \mid \langle t\|VT\rangle \in r \wedge c(\langle t\|VT\rangle)\} \\
\pi_f^{vt}(r) & \triangleq & \{\langle t_1\|VT\rangle \mid \langle t_2\|VT\rangle \in r \wedge t_1 = \langle f(t_2)\|VT\rangle\} \\
r_1 \cup^{vt} r_2 & \triangleq & \{\langle t\|VT\rangle \mid \langle t\|VT\rangle \in r_1 \vee \langle t\|VT\rangle \in r_2\} \\
r_1 \bowtie_c^{vt} r_2 & \triangleq & \{\langle\langle t_1, VT_1\rangle \circ \langle t_2, VT_2\rangle\|VT\rangle \mid \langle t_1\|VT_1\rangle \in r_1 \wedge \langle t_2\|VT_2\rangle \in r_2 \wedge \\
& & \quad c(\langle t_1, VT_1\rangle \circ \langle t_2, VT_2\rangle) \wedge \\
& & \quad VT = \mathit{intersect}(VT_1, VT_2) \wedge VT_1 \mathit{\ overlaps\ } VT_2\} \\
r_1 \setminus^{vt} r_2 & \triangleq & \{\langle t\|VT\rangle \mid \langle t\|VT_1\rangle \in r_1 \wedge \\
& & \quad (\exists VT_2(\langle t\|VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\
& & \quad (\exists VT_3(\langle t\|VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\
& & \quad VT^- < VT^+ \wedge \\
& & \quad \neg\exists VT_4(\langle t\|VT_4\rangle \in r_2 \ \wedge VT_4^+ > VT^- \ \wedge VT_4^- < VT^+)\} \\
AG_{agg,f}^{vt}(r) & \triangleq & \{\langle t \circ a\|VT\rangle \mid \langle t\|VT_1\rangle \in r \ \wedge \langle t_2\|VT_2\rangle \in r \ \wedge f(t) = f(t_2) \wedge \\
& & \quad ((VT^- = VT_1^- \ \wedge VT^+ = VT_2^-) \vee \\
& & \qquad (VT^- = VT_1^- \ \wedge VT^+ = VT_2^+) \vee \\
& & \qquad (VT^- = VT_1^+ \ \wedge VT^+ = VT_2^+)) \ \wedge VT^- < VT^+ \wedge \\
& & \quad \neg\exists\langle t_4\|VT_4\rangle \in r(f(t) = f(t_4) \wedge \\
& & \qquad\qquad ((VT^- < VT_4^- < VT^+) \vee (VT^- < VT_4^+ < VT^+))) \wedge \\
& & \quad a = agg(\{t_3 | \langle t_3\|VT_3\rangle \in r \ \wedge VT_3 \mathit{\ overlaps\ } VT \wedge f(t) = f(t_3)\})\}
\end{array}
$$

Figure 7: Semantics of the temporal algebra

Assume that `p` is a table without temporal support and `q` is a valid-time table. To evaluate this query according to temporal upward compatibility, we timeslice `q` as of now.

$$[\![Q]\!]_{\mathrm{SQL/T}}(p, q) = [\![Q]\!]_{\mathrm{standard}}(p, \tau_{now}^{vt}(q)) = \pi_{p.X}(p \bowtie_{p.X=q.X} \tau_{now}^{vt}(q)) \ .$$

Now consider the SQL/Temporal query $Q_2 =$ **NONSEQUENCED VALIDTIME** Q. `q` is converted to a table without temporal support with an additional column, containing the valid-time period of the row, then the query is evaluated according to the standard SQL semantics.

$$
\begin{array}{lll}
[\![Q_2]\!]_{\mathrm{SQL/T}}(p, q) & = & [\![\texttt{NONSEQUENCED VALIDTIME } Q]\!]_{\mathrm{SQL/T}}(p, q) \\
& = & [\![Q]\!]_{\mathrm{standard}}(p, SN(q)) \\
& = & \pi_{p.X}(p \bowtie_{p.X=q.X} SN(q)) \ .
\end{array}
$$

Completing this example, let's examine sequenced valid semantics. Assume that `p` is also a table with valid-time support (sequenced semantics requires all underlying tables to be tables with valid-time support). Consider $Q_3 =$ **VALIDTIME** Q.

$$
\begin{array}{lll}
[\![Q_3]\!]_{\mathrm{SQL/T}}(p, q) & = & [\![\texttt{VALIDTIME } Q]\!]_{\mathrm{SQL/T}}(p, q) \\
& = & [\![Q]\!]_{\mathrm{temporal}}(p, q) \\
& = & \pi_{p.X}^{vt}(p \bowtie_{p.X=q.X}^{vt} q) \ .
\end{array}
$$

Note that apart from the ^{vt}-superscripts, which are added to relational algebra operators, the translation between SQL queries and relational algebra expressions for all three types of queries has not changed at all. □

The semantics of **VALIDTIME**(c) are quite simple. If c is associated with a row $\langle t \mid\mid VT\rangle$ of a table with valid-time support, then

$$[\![\texttt{VALIDTIME}(c)]\!] = VT \ .$$

If a period expression follows VALIDTIME, the result of the select statement is restricted to the period specified. This can be accomplished by an extension of the timeslice operator τ to take a period expression as its subscript.

$$[\![\texttt{VALIDTIME}\ p\ \text{<query expression>}]\!]_{\mathrm{SQL/T}}(r_1,\ldots,r_n)$$
$$\triangleq \{\langle t \,||\, VT\rangle \mid \langle t \,||\, VT'\rangle \in [\![\texttt{VALIDTIME}\ \text{<query expression>}]\!]_{\mathrm{SQL/T}}(r_1,\ldots,r_n) \wedge VT = VT' \cap [\![p]\!] \wedge VT \neq \emptyset\}$$

If a period expression follows NONSEQUENCED VALIDTIME, a valid-time table results, with the valid-time period as specified. Assume in the following that all the tables are tables with valid-time support.

$$[\![\texttt{NONSEQUENCED VALIDTIME}\ p\ \text{<query expression>}]\!]_{\mathrm{SQL/T}}(r_1,\ldots,r_n)$$
$$= \{\langle t \,||\, [\![p]\!]\rangle \mid t \in ([\![\texttt{NONSEQUENCEDVALIDTIME}\ \text{<query expression>}]\!]_{\mathrm{SQL/T}}(r_1,\ldots,r_n)\}$$

8 A Foundation for Implementing the Extensions

We first provide a mapping of temporal relational operations to conventional relational algebra expressions. We then list a range of alternatives for implementing the temporal relational operators.

8.1 Implementing the Temporal Algebra

Here, we give the conventional algebraic equivalents for the temporal algebraic operators. We emphasize that conventional operators range over a different domain (tables without temporal support) than do temporal operators (tables with valid-time support). In Figure 8, the set A_{r_i} contains the explicit attributes of table r_i, and a is the attribute appended by AG.

A "vt" superscript on a table indicates that it is a table with valid-time support; those tables without such a superscript are tables without temporal support, each with an explicit VT column. The auxiliary function $SN(r) = \{\langle t, VT\rangle \mid \langle t \| VT\rangle \in r\}$ maps a table with valid-time support into a table without temporal support with valid-time being an explicit attribute. It is assumed that tables with valid-time support are mapped into tables without temporal support using SN before conventional algebraic operators are applied. Note that function SN is not needed at the implementation level. However, it is required here because Codd's relational algebra operators are only well-defined over tables without temporal support. Finally, there is a rename operator, $\rho_i(r)$ that gives table r the name i. Again, the aggregate operator is the most complex. The relational difference and the outer Cartesian product are the analogs of "$\neg\exists$" in the calculus; the inner Cartesian product and unions (to compute t_2) are the analogs of the two row variables in the calculus.

Example 12: We continue by mapping the temporal algebraic equivalent of the SQL/Temporal query $Q_2 = \texttt{VALIDTIME}\ Q_1$ into the snapshot algebra. Here, we assume that the table p has a single column, X, and that the table q has two columns, X and Y.

$$\begin{aligned}
[\![Q_2]\!]_{\mathrm{SQL/T}}(p,q) &= [\![\texttt{VALIDTIME}\ Q_1]\!]_{\mathrm{SQL/T}}(p,q) = [\![Q_1]\!]_{\mathrm{temporal}}(p,q) = \pi^{vt}_{p.X}(p \bowtie^{vt}_{p.X=q.X} q)\\
&= \pi_{X_p,VT}(\pi_{X_p,VT_p,X_q,Y_q,VT_q,VT=intersect(VT_p,VT_q)}(SN(p) \bowtie_{p.X=q.X \wedge VT_p\ overlaps\ VT_q} SN(q)))\\
&= \pi_{X_p,VT=intersect(VT_p,VT_q)}(SN(p) \bowtie_{p.X=q.X \wedge VT_p\ overlaps\ VT_q} SN(q))
\end{aligned}$$

□

8.2 Alternatives for Implementing SQL/Temporal

The transformations from the temporal algebra to the conventional algebra gives us several options for implementing SQL/Temporal.

1. Map temporal queries into temporal algebra, then into regular algebra, according to Figure 8, then back into SQL.

$$
\begin{array}{lcl}
\sigma_c^{vt}(r^{vt}) & \leadsto & \sigma_c(r) \\
\pi_f^{vt}(r^{vt}) & \leadsto & \pi_{f,VT}(r) \\
r_1^{vt} \cup^{vt} r_2^{vt} & \leadsto & r_1 \cup r_2 \\
r_1^{vt} \bowtie_c^{vt} r_2^{vt} & \leadsto & \pi_{A_{r_1},VT_{r_1},A_{r_2},VT_{r_2},VT=intersect(VT_{r_1},VT_{r_2})}(r_1 \bowtie_{c \wedge VT_{r_1} \text{ overlaps } VT_{r_2}} r_2) \\
r_1^{vt} \setminus^{vt} r_2^{vt} & \leadsto & t_2 \setminus \pi_{A_{t_2},VT_{t_2}}(t_2 \bowtie_{A_{t_2}=A_{r_2} \wedge VT_{t_2} \text{ overlaps } VT_{r_2}} r_2) \\
 & & t_2 = t_1 \cup \pi_{A_{r_1},period(VT^+_{r_2},VT^-_{t_1})}(t_1 \bowtie_{A_{t_1}=A_{r_2} \wedge VT^-_{t_1} \leq VT^+_{r_2} \wedge VT^+_{r_2} < VT^+_{t_1}} r_2) \\
 & & t_1 = r_1 \cup \pi_{A_{r_1},period(VT^-_{r_1},VT^-_{r_2})}(r_1 \bowtie_{A_{r_1}=A_{r_2} \wedge VT^-_{r_1} < VT^-_{r_2} \wedge VT^-_{r_2} \leq VT^+_{r_1}} r_2) \\
AG^{vt}_{agg,f}(r^{vt}) & \leadsto & AG_{agg,f}(t_2 - \pi_{A_1,A_2,a,VT_1^-,VT_1^+}(\\
 & & \quad \sigma_{f(A_1)=f(A_2) \ \wedge((VT_1^- < VT_2^- < VT_1^+) \vee (VT_1^- < VT_2^+ < VT_1^+))}(\rho_1(t_2) \times \rho_2(r)))) \\
 & & t_2 = \pi_{A_1,a,period(VT_1^-,VT_2^-)}(t_1) \cup \pi_{A_1,a,period(VT_1^-,VT_2^+)}(t_1) \cup \pi_{A_1,a,period(VT_1^+,VT_2^+)}(t_1) \\
 & & t_1 = \pi_{A_1,a,VT_1,VT_2}(\sigma_{f(A_1)=f(A_2)}(\rho_1(r)) \times \rho_2(r))
\end{array}
$$

Figure 8: Snapshot equivalents of the the temporal algebra operators

2. Map temporal queries into temporal algebra, then, according to Figure 7, directly into SQL.
3. Map temporal queries directly into SQL, utilizing the temporal algebra implicitly in the query rewrite phase (this is what the prototype does).

EXAMPLE 13: Continuing with the previous example, the SQL/Temporal query Q_2 on tables with valid-time support can be mapped to an SQL3 query on tables without temporal support where the implicit timestamps are now placed in explicit attributes.

```
SELECT p.X, p.VT INTERSECT_P q.VT AS VT
FROM p, q
WHERE p.X = q.X AND p.VT OVERLAPS q.VT
```

Here, we use the `OVERLAPS` predicate and the `INTERSECT_P` operator already present in SQL/Temporal. □

Finally, we point out some possibilities for query optimization.

1. Map temporal queries into temporal algebra, optimize as with SQL algebra (with existing transformations and/or new cost formulas), then map back in SQL.
2. Map temporal queries into temporal algebra, then into SQL algebra, then optimize, then evaluate.
3. Map temporal queries into temporal algebra, then optimize (again, using new cost formulas), then evaluate the temporal algebra directly, with the concomitant increase in performance.

8.3 Implementing Temporal Assertions and Constraints

The general approach to checking an assertion is to negate it and then execute it as a query [2]. If the query result is empty, i.e., if no rows are returned, the assertion is respected, otherwise it is violated.

EXAMPLE 14: To check the assertion `emp_has_sal` from Section 6.1.2 we execute the query

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

A non-empty result indicates a violation of the assertion. □

Temporal assertions and constraints, specified with `VALIDTIME`, can be checked in a similar way, with a `VALIDTIME SELECT` statement.

First, note that database systems have to improve the sketched mechanism to achieve acceptable performance. Well-known techniques include incremental consistency checking, simplification of assertions, and special-purpose checking algorithms for, e.g., column constraints. Second, it becomes obvious how important it is to address *all* aspects of a query language when transitioning from a nontemporal to a temporal database system. Negation, which might be used rarely in queries asked by users, is crucial for answering assertions because these usually involve some form of implication, i.e., involve negation. In our approach, it is no harder to state a temporal negation than it is to state a temporal join. This makes specification (and implementation) of assertions particularly elegant.

8.4 Implementing Nonsequenced Semantics

Interestingly, supporting nonsequenced semantics is much easier than supporting sequenced semantics.

Assume as before that a table with valid-time support is *represented* at the physical level as a conventional table with an additional column (of an implementation-dependent name) containing the valid time. Then a nonsequenced query can be evaluated as before, with no special treatment. `VALIDTIME()` simply returns the value of the new column. The optional period expression is treated as an additional column in the evaluation of a nonsequenced query.

A major advantage of nonsequenced semantics is its ease of implementation. The disadvantage of nonsequenced semantics is that the user is responsible for handling the time dimension explicitly, which is why *sequenced* semantics is so important.

9 Summary

In this change proposal, 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` reserved words, to indicate valid-time support (in the case of schema specification statements) and sequenced semantics (in the case of queries, modifications, views, 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. We provided 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 provided the semantics for nonsequenced queries. Finally, we listed alternative implementation approaches which vary in the degree of implementation difficulty and the achievable performance efficiency, and showed that implementing nonsequenced semantics is straightforward.

Appendix A provides formal definitions of the properties discussed in Section 5.

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 two new reserved words, `NONSEQUENCED` and `VALIDTIME`, are required.
- Satisfaction of temporal upward compatibility ensures that existing applications do not break when tables without temporal support have such support added.
- Satisfaction of sequenced semantics ensures that temporal queries, modifications, views, assertions, and constraints are easy to specify, formalize, and implement.
- Nonsequenced semantics permits tables with valid-time support to be converted to tables without such support, with an explicit timestamp column, and such for valid-time support to be added to tables, even within a query.

- Since the semantics is defined in terms of the non-temporal semantics, the extensions are compatible with *all* the facilities of SQL3.
- A simple period expression permits the temporal scope to be specified.
- A prototype implementation exists [14]; this prototype was invaluable in refining the language additions.
- Transaction time support will require few syntactic or semantic extensions, and will be fully compatible and consistent with these valid-time features.

10 Proposed Language Extensions

The syntax is given as extensions to "Database Language SQL — Part 7: Temporal" [8].

11 Clause 3 Definitions, notations, and conventions

11.1 Subclause 3.1 Definitions

1) Add the following terms.

g) **row with valid-time support**: A row with valid-time support is a row with an associated valid time, which is a value of data type <period type>.

h) **valid time of a row with valid-time support**: The valid time of a row with valid-time support is the period during which the values in the columns of the row are known to be valid.

Note to proposal reader: The valid-time period is not required to be maximal.

i) **table with valid-time support**: A table with valid-time support is one in which each row is a row with valid-time support.

j) **valid-time state of a table with valid-time support at a valid time**: The valid-time state of a table with valid-time support TV at a specified valid time T is the table without valid-time support comprising the rows of TV associated with valid-time periods that overlap T.

k) **current valid-time state of a table with valid-time support**: The current valid-time state of a table with valid-time support is the valid-time state of that table at valid time CURRENT_TIMESTAMP.

l) **precision of a table with valid-time support**: The precision of a table with valid-time support is the precision of the valid-time period of its rows.

m) **forever**: Forever is the maximum datetime value, 9999-12-31 23:59:59.999999....

12 Clause 4 Concepts

1) Insert the following Subclause, "Tables", to SQL/Temporal immediately following Subclause 4.2.3, "Period predicates".

12.1 Subclause 4.3 Tables

Every table descriptor also includes:

- An indication of whether the table has valid-time support or does not have valid-time support.
- The valid-time precision of the table, if the table has valid-time support.

2) Insert the following Subclause, "Integrity constraints", to SQL/Temporal immediately following Subclause 4.3, "Tables".

12.2 Subclause 4.4 Integrity constraints

Every constraint descriptor also includes:

- An indication of whether the constraint is specified without VALIDTIME, with VALIDTIME but without NONSEQUENCED, or with NONSEQUENCED VALIDTIME.
- The valid-time period, if any, associated with the constraint.

1) Insert the following Subclause, "Meaning of statements on tables with valid-time support", to SQL/Temporal immediately following Subclause 4.4, "Integrity constraints".

12.3 Subclause 4.5 Meaning of statements on tables with temporal support

Temporal upward compatible queries (i.e., SELECT without VALIDTIME) treat each underlying table that has valid-time support as a table without valid-time support, by using instead the current valid-time state of the table. Hence, a query evaluated with temporal upward compatibility on a table with valid-time support will use only the current valid-time state in the evaluation.

Sequenced valid queries (i.e., VALIDTIME SELECT) apply only on tables with valid-time support, and result in tables with valid-time support. The meaning of sequenced valid queries is defined in terms of the meaning of queries on tables without valid-time support. Let Q be a sequenced valid query, with Q = VALIDTIME Q1, where Q1 is a query without VALIDTIME. The meaning of Q1 on tables without temporal support is already defined by this International Standard. Let R be the valid-time table that is the result of Q on one or more tables with valid-time support. For all times T, the state of R at time T is the result of Q1 according to the General Rules in Subclause 7.4, "<query expression>" on the states of the underlying tables at time T. Any R that satisfies this property is a valid result of Q.

Nonsequenced valid queries (i.e., NONSEQUENCED VALIDTIME SELECT) treat each underlying table that has valid-time support as a table without valid-time support, but with an additional unnamed column whose value for a row in the table is the valid-time period associated with the corresponding row in the original table. With this substitution, the General Rules in Subclause 7.4, "<query expression>" apply.

The <value expression> following VALIDTIME is used in two ways. Within a nonsequenced valid query, it supplies the valid-time period of the computed rows. In a sequenced valid query, it specifies the "temporal scope" of the query: the result is computed with sequenced valid semantics, then the valid-time periods of the result are intersected with the value of the <value expression> to determine the final valid-time period.

These same concepts (temporal upward compatibility, sequenced valid, and nonsequenced valid) also apply to integrity constraints, assertions, views, cursors and modification statements.

Note to proposal reader: Very informally, what is going on is that the SQL3 semantics of a query Q is treated as a black box. Put a query and a database (conventional, without valid-time support) in one side, and out comes a (carefully specified) table (without valid-time support) on the other side.

The semantics of the nonsequenced, sequenced, and temporally upward compatible queries, modifications, etc., are specified by using this black box.

The advantages are numerous. (1) We don't have to modify each page of the specification; instead, the additions to the specification are small and isolated. (2) As SQL3 grows, with new constructs, the temporal query variants still work fine. (3) The intuition of the user is aided by the fact that a temporal version of a query is defined in terms of the original, nontemporal version of the query.
End of note.

13 Clause 5 Lexical elements

13.1 Subclause 5.1 <token> and <separator>

1) In the Format, add the following two new alternatives to <reserved word>:

| NONSEQUENCED
| VALIDTIME

14 Clause 6 Scalar expressions

1) Insert the following Subclause, "<item reference>", to SQL/Temporal immediately preceding Subclause 6.2, "<set function specification>".

14.1 Subclause 6.1 <item reference>

Function

Reference a column, parameter, or variable.

Format

No additional Format items.

Syntax Rules

1. (Insert this SR) If a <column name> CN is contained in a <value expression> that is simply contained in a <valid option> that is simply contained in a <query expression> QE, then the scope clause of CN is the <query expression body> that is simply contained in QE.
2. (Insert this SR) If a <column name> CN is contained in a <value expression> that is simply contained in a <valid option> that is simply contained in a <select statement: single row> SS, then the scope clause of CN is the <select list> that is simply contained in SS.
3. (Insert this SR) If a <column name> CN is contained in a <value expression> that is simply contained in a <valid option> that is simply contained in a <delete statement: positioned> or a <delete statement: searched> or an <update statement: searched> or an <update statement: positioned>, then the scope clause of CN is the <table reference> that is simply contained in the statement.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

14.2 Subclause 6.5 <period value expression>

1) In the Format, add the following new alternative to <period primary>:

| <valid function>

2) In the Format, add the following BNF production:

<valid function> ::=
VALIDTIME <left paren> <item qualifier> <right paren>

Note to proposal reader: An <item qualifier> is either a <table name> or a <correlation name>.

3) Insert the following two Syntax Rules:

1. (Insert this SR) The <item qualifier> of a <valid function> shall be associated with a table T with valid-time support of precision P.
2. (Insert this SR) The data type of <valid function> shall be <period type>, with a precision of P.

Note to proposal reader: The precision of a table with valid-time support was defined in Subclause 3.1, "Definitions".

4) Insert the following General Rule:

1. (Insert this GR) When a <valid function> VF is evaluated for a row R of T, the value of VF is the valid-time period of R.

Language opportunity: It would be helpful if this function were also available in PSM to apply to values of type ROW.

15 Clause 7 Query expressions

1) Insert the following new Subclause, "<query expression>", to SQL/Temporal immediately following Subclause 7.3, "<period value constructor>".

15.1 Subclause 7.4 <query expression>

Function

Specify a table.

Format

<query expression> ::=
  [<with clause>] <temporal query expression body>

<temporal query expression body> ::=
  [<time option>] <query expression body>

Note to proposal reader: This adds an optional <time option> to <query expression>.

<time option> ::=
  <valid option>

<valid option> ::=
    [NONSEQUENCED] VALIDTIME [<value expression>]

Note to proposal reader: The <time option> is expanded in the transaction-time change proposal, in a similar fashion to <valid option> proposed here. There are five cases:

1. `SELECT`
   - works on anything
   - evaluates to a table with no temporal support
2. `VALIDTIME SELECT`
   - works only on tables with valid-time support
   - evaluates to a table with valid-time support
3. `VALIDTIME` <period exp> `SELECT`
   - like `VALIDTIME SELECT`, but only returns timestamps within <period exp> (a simple example is `VALIDTIME PERIOD '[1995-01-01 - 1995-12-31]' SELECT`)
4. `NONSEQUENCED VALIDTIME SELECT`
   - works on anything
   - acts like tables with valid-time support have an explicit timestamp column
   - evaluates to a table with no temporal support
5. `NONSEQUENCED VALIDTIME` <period exp> `SELECT`
   - like `NONSEQUENCED VALIDTIME SELECT`, but uses the <period exp> as a timestamp, and thus returns a table with valid-time support

To convert a table with valid-time support to a table with no temporal support, use `SELECT` (if only the current state is of interest) or `NONSEQUENCED VALIDTIME SELECT`.

To convert a table without valid-time support to a table with valid-time support, use `NONSEQUENCED VALIDTIME` <period exp> `SELECT`.

End of note.

Syntax Rules

1. (Replace SR1b) For all i between 1 and n, the scope of the <query name> WQN that is immediately contained in WLEi is the <query expression> that is immediately contained in every <with list element> WLEk, where k ranges from i+1 to n, and the <temporal query expression body> that is immediately contained in <query expression>. A <table or query name> that is contained in this scope that immediately contains WQN is a query name in scope.

   Note to proposal reader: This simply replaces <query expression body> with <temporal query expression body>.

2. (Add to SR5) d) <time option> is not specified in the <temporal query expression body> that is contained in <query expression>.

   Note to proposal reader: This specifies that if <time option> is specified, the result of <query expression> is not inherently updatable. See the language opportunity, below.

3. (Insert this SR) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in the <time option> that is simply contained in <query expression>, then each exposed table, query, or correlation name that is contained in the <query expression body> without an intervening <from clause> shall identify a table with valid-time support and with identical precision P.

   Note to proposal reader: This ensures that sequenced valid queries are only evaluated "over" tables with valid-time support.

4. (Insert this SR) If VALIDTIME is specified in the <valid option> of a <query expression> Q, then either Q shall be simply contained in a <from clause> or Q shall be the outermost <query expression>.

   Note to proposal reader: VALIDTIME is allowed in only two places: prepended to the outermost <query expression>, or immediately within a <from clause>. The reason is that the <time option> applies to the entire <query expression>, evaluated on the exposed tables, queries, and correlation names, which are specified in the <from clause>s of the query.

5. (Insert this SR) The data type of the <value expression> that is contained in the <valid option> that is contained in <time option> shall be <period type>.

   NOTE 6 - Subclause 6.3, "<item reference>" restricts the scope of column names in <value expression>.

6. (Insert this SR) Let T be the result of the <query expression>.

   Case:

   a) If VALIDTIME is specified and NONSEQUENCED is not specified in <valid option>, then T shall be a table with valid-time support and with precision P. The precision of the <value expression> that is contained in the <valid option> of <time option> shall be P.

   b) If NONSEQUENCED VALIDTIME is specified in <time option>, then

      Case:

i) If <value expression> is specified in the <valid option> of <time option>, then T shall be a table with valid-time support and with a precision of that of <value expression>.

ii) Otherwise, T shall be a table without valid-time support.

c) Otherwise, T shall be a table without valid-time support.

Note to proposal reader: If NONSEQUENCED VALIDTIME is specified, the precision of the <value expression> of <valid option> is arbitrary.

Access Rules

No additional Access Rules.

General Rules

1. (Replace GR1a) For every <with list element> WLE, let WQN be the <query name> immediately contained in WLE. Let WQE be the <temporal query body> immediately contained in WLE. Let WLT be the table resulting from evaluation of WQE, with each column name replaced by the corresponding element of the <with column list>, if any, immediately contained in WLE.

   Note to proposal reader: This simply replaces <query expression> with <temporal query expression body>.

2. (Insert this GR) Case:

   a) If VALIDTIME is specified and NONSEQUENCED is not specified in <valid option>, then the result of <temporal query expression body> TQEB at each valid time granule T of precision P is the result of the <query expression body> of TQEB with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with its state at valid time T. If <value expression> VE is specified in the <valid option> that is contained in <time option>, then for each row R resulting from the initial evaluation of TQEB,
   Case:

   i) If the value of VE and the valid-time period VP of R overlap, then the resulting valid-time period of R is the result of P_INTERSECT on the value of VE and VP.

   ii) Otherwise, R is not included in the final result of TQEB.

   Note to proposal reader: P_INTERSECT is intersection on periods, as defined in Subclause 6.5, "<period value expression>", in SQL/Temporal.

   b) If NONSEQUENCED VALIDTIME is specified in <time option>, then the result of <temporal query expression body> TQEB is the result of the <query expression body> of TQEB with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with a table with no valid-time support with rows with identical values for the columns. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is implementation-dependent, whose data type is a <period type> with a precision of that of the valid-time period of DT, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original valid-time period of the corresponding row in DT. If <value expression> is specified in the <valid option> of <time option>, then the valid-time period of the row of the result of TQEB is the value of <value expression>.

   c) Otherwise, the result of <temporal query expression body> TQEB is the result of the <query expression body> of TQEB with each of its leaf generally underlying tables with valid-time support with no intervening <from clause> replaced with its current valid-time state.

Language opportunity: It may be possible to allow temporal query expressions to be updatable.

Language opportunity: It would be nice if <value expression> that is contained in the <valid option> that is contained in <time option> also be allowed to be of a datetime data type, interpreted as a period containing one granule. This would allow statements of the form `VALIDTIME DATE '1996-01-01' SELECT`.

16 Clause 10 Schema definition and manipulation

1) Insert this new Subclause, "<table definition>", to SQL/Temporal immediately following Subclause 10.1, "<default clause>".

16.1 Subclause 10.2 <table definition>

Function

Define a persistent base table, a created local temporary table, or a global temporary table.

Format

```
<table definition> ::=
        CREATE [ <table scope> ] TABLE <table name>
            { <table element list> |<subtable clause> [ <table element list> ] }
            [ <temporal definition> ]
            [ ON COMMIT <table commit action> ROWS ]
```

Note to proposal reader: This augments the production for the non-terminal <table definition> with an additional, optional clause to specify that the new table is to be a table with valid-time support.

```
<temporal definition> ::=
        AS VALIDTIME [ <table precision> ]

<table precision> ::=
        <left paren> <end field> <right paren>
```

Note to proposal reader: <end field> is YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND, with SECOND allowing an optional fractional seconds precision.

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

1. (Add to GR3)

   f) Whether the table has valid-time support or does not have valid-time support.

   g) If the table has valid-time support, then the valid-time precision of the table.

   Note to proposal reader: These two items are added to the table descriptor.

2. (Insert this GR) If <temporal definition> is specified, then the descriptor for the table indicates that the table has valid-time support with a precision of <table precision>.

   Note to proposal reader: Otherwise, the table does not have valid-time support.

16.2 Subclause 10.3 <column definition>

2) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.2, "<table definition>".

Function

Define a column of a table.

Format

<column constraint definition> ::=
[<constraint name definition>] <temporal column constraint>

<temporal column constraint> ::=
[<time option>] <column constraint> [<constraint attributes>]

Note to proposal reader: This adds an optional <time option> to column constraints.

Syntax Rules

1. (Insert this SR) If VALIDTIME is specified in <valid option> contained in <time option>, then T shall be a table with valid-time support.
2. (Insert this SR) If VALIDTIME is specified and NONSEQUENCED is not specified in <valid option> that is contained in <time option>,

   Case:

   a) If <column constraint> is <references specification>, then the table identified by <table name> that is simply contained in the <referenced table and columns> of <references specification> shall be a table with valid-time support.

   b) If <column constraint> is <check constraint definition>, then each table associated with an exposed <table name>, <query expression>, or <correlation name> that is contained in the <column constraint> without an intervening <from clause> shall be a table with valid-time support and with identical precision.
3. (Insert this SR) The precision of <value expression> VE that is contained in the <valid option> of <time option> shall be the precision of T. VE shall be a <literal>.
4. (Insert this SR) If NONSEQUENCED VALIDTIME is specified in the <time option> that is contained in <column constraint definition> , then VO shall not contain <value expression>.
5. (Insert this SR) If <time option> is specified, then the <column constraint> shall not be NOT NULL.

   Note to proposal reader: This restriction might be lifted in the future, along with various other schema manipulation restrictions. See the language opportunity, below.
6. (Insert this SR) The <temporal column constraint> TCC is equivalent to a <temporal table constraint> with a <time option> of the <time option> of TCC and a <table constraint> that is equivalent to the <column constraint> that is contained in TCC.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Language opportunity: It may be possible to allow <time option> with NOT NULL, but the implications on nullability and functional dependencies should be carefully considered.

16.3 Subclause 10.4 <table constraint definition>

3) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.3, "<column definition>".

Function

Specify an integrity constraint.

Format

<table constraint definition> ::=
    [<constraint name definition>] [<time option>] <temporal table constraint>

<temporal table constraint> ::=
    <table constraint> [<constraint attributes>]

Note to proposal reader: This adds an optional <time option>. For constraints and assertions, there are four cases:

1. `CHECK`
   - works on anything
   - only considers current state
2. `VALIDTIME CHECK`
   - works only on tables with valid-time support
   - the constraint/assertion must be true for the state at every valid time
3. `VALIDTIME` <period exp> `CHECK`
   - like `VALIDTIME CHECK`, but only considers the times in <period exp> (a simple example is `VALIDTIME PERIOD '[1995-01-01 - 1995-12-31]' CHECK`)
4. `NONSEQUENCED VALIDTIME CHECK`
   - works on anything
   - acts like tables with valid-time support have an explicit timestamp column; all rows are considered at once

`NONSEQUENCED VALIDTIME` <period exp> `CHECK` is not allowed.
End of note.

Syntax Rules

1. (Insert this SR) Let T be the table defined by the <table definition> containing this <table constraint definition>.
2. (Insert this SR) If VALIDTIME is specified in the <valid option> that is contained in <time option>, then T shall be a table with valid-time support with precision P. The precision of <value expression> VE that is contained in the <valid option> that is contained in <time option> shall be P. VE shall be a <literal>.
3. (Insert this SR) If VALIDTIME is specified and NONSEQUENCED is not specified in <table constraint definition>, then each exposed table, query, or correlation name that is contained in the <table constraint> without an intervening <from clause> shall identify a table with valid-time support and with identical valid-time period precision.

4. (Insert this SR) If <valid option> VO that is contained in <column constraint definition> contains NONSEQUENCED, then VO shall not contain <value expression>.

Access Rules

No additional Access Rules.

General Rules

1. (Append to GR2) The table constraint descriptor includes an indication of whether the constraint has valid-time support or does not have valid-time support, as well as the valid-time period, if any, of the table constraint, if the table constraint has valid-time support.
2. (Insert this GR) Case:
   a) If VALIDTIME is specified and NONSEQUENCED is not specified in <valid option>, then Case:
      i) If <value expression> V is contained in the <valid option> of <time option>, then <temporal table constraint> is satisfied if the contained <table constraint> is satisfied for each time granule TG of the value of V, with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with its state at valid time TG.
      ii) Otherwise, <temporal table constraint> is satisfied if the contained <table constraint> is satisfied for each time granule TG of precision P, with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with its state at valid time TG.
   b) If NONSEQUENCED VALIDTIME is specified in <time option>, then <temporal table constraint> is satisfied if the contained <table constraint> is satisfied when each leaf generally underlying table with valid-time support with no intervening <from clause> is replaced with a table with no valid-time support with rows with identical values for the columns. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is implementation-dependent, whose data type is a <period type> with a precision of that of the valid-time period of DT, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original valid-time period of the corresponding row in DT.
   c) Otherwise, <temporal table constraint> is satisfied if the contained <table constraint> is satisfied when each of its leaf generally underlying tables with valid-time support with no intervening <from clause> is replaced with its current valid-time state.

16.4 Subclause 10.5 <alter table statement>

4) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.4, "<table constraint definition>".

Function

Change the definition of a table.

Format

<alter table action> ::=
    !! All alternatives from ISO/EIC 9075
    | <add valid definition>
    | <drop valid definition>
    | <convert valid definition>

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

16.5 Subclause 10.6 <add valid definition>

5) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.5, "<alter table statement>".

Function

Add valid-time support to a table.

Format

<add valid definition> ::=
    ADD VALIDTIME [<table precision>]

Syntax Rules

1. (Insert this SR) Let T be the table identified by the <table name> that is immediately contained in the <alter table statement> that immediately contains <add valid definition>.
2. (Insert this SR) T shall be a table without valid-time support.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Valid-time support is added to each row of T, by associating with that row a valid-time period from the current timestamp to forever with a precision of <table precision>. The descriptor of T is altered to indicate that T has valid-time support, of the precision of the <end field> that is contained in <table precision>.

16.6 Subclause 10.7 <drop valid definition>

6) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.6, "<add valid definition>".

Function

Drop valid-time support from a table.

Format

```
<drop valid definition> ::=
        DROP VALIDTIME
```

Syntax Rules

1. (Insert this SR) Let T be the table identified by the <table name> that is immediately contained in the <alter table statement> that immediately contains <drop valid definition>.
2. (Insert this SR) T shall be a table with valid-time support.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Valid-time support is removed from T, by replacing T with the result of
   SELECT * FROM T

   Note to proposal reader: That is, only the current valid-time state is retained.

   The descriptor of T is altered to indicate that T does not have valid-time support.

16.7 Subclause 10.8 <convert valid definition>

7) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.7, "<drop valid definition>".

Function

Change the valid-time precision of a table.

Format

```
<convert valid definition> ::=
        ALTER VALIDTIME WITH PRECISION <table precision>
```

Syntax Rules

1. (Insert this SR) Let T be the table identified by the <table name> that is immediately contained in the <alter table statement> that immediately contains <convert valid definition>.

2. (Insert this SR) T shall be a table with valid-time support.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) T is converted to the new precision specified effectively by the execution of the following statements. Let TC be the columns of T, and let P be the <end field> of <table precision>. The descriptor of T is altered to indicate that T has a valid-time precision of P.

```
CREATE TABLE Temp ( TC ) AS VALIDTIME P

INSERT INTO Temp
VALIDTIME SELECT ( TC )
FROM (NONSEQUENCED VALIDTIME CAST( when AS PERIOD(TIMESTAMP P ))
        SELECT *, VALIDTIME(T) AS when
        FROM T) AS S

DROP TABLE T

CREATE TABLE T ( TC ) AS VALIDTIME P

INSERT INTO T
        VALIDTIME SELECT * FROM Temp

DROP TABLE Temp
```

16.8 Subclause 10.9 <assertion definition>

8) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.8, "<convert valid definition>".

Function

Specify an integrity constraint by means of an assertion and specify when the assertion is to be checked.

Format

```
<triggered assertion> ::=
        [ <time option> ]
            CHECK <left paren> <search condition> <right paren>
```

Note to proposal reader: This adds an optional <time option>.

Syntax Rules

1. (Insert this SR) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in <time option>, then each exposed table, query, or correlation name that is contained in the <search condition> without an intervening <from clause> shall identify a table with valid-time support of precision P. The precision of the <value expression> that is contained in the <valid option> that is contained in <time option> shall be P. VE shall be a <literal>.

2. (Insert this SR) If NONSEQUENCED VALIDTIME is specified in the <valid option> VO contained in the <time option> TO of <triggered assertion>, then VO shall not contain <value expression>.

Access Rules

No additional Access Rules.

General Rules

1. (Append to GR4) The assertion descriptor includes an indication of whether the assertion has valid-time support or does not have valid-time support, as well as the valid-time period, if any, of the assertion, if the assertion has valid-time support.

2. (Insert this GR) Case:

   a) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in <time option>, then
   Case:

      i) If <value expression> V is contained in the <valid option> that is contained in <time option>, then <triggered assertion> is satisfied if the contained <search condition> is satisfied for each time granule TG of the value of V, with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with its state at valid time TG.

      ii) Otherwise, <triggered assertion> is satisfied if the contained <search condition> is satisfied for each time granule TG of precision P, with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with its state at valid time TG.

   b) If NONSEQUENCED VALIDTIME is specified in <time option>, then <triggered assertion> is satisfied if the contained <search condition> is satisfied when each leaf generally underlying table with valid-time support with no intervening <from clause> is replaced with a table with no valid-time support with rows with identical values for the columns. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is implementation-dependent, whose data type is a <period type> with a precision of that of the valid-time period of DT, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original valid-time period of the corresponding row in DT.

c) Otherwise, <triggered assertion> is satisfied if the contained <search condition> is satisfied when each of its leaf generally underlying tables with valid-time support with no intervening <from clause> is replaced with its current valid-time state.

17 Clause 11 Period manipulation rules

1) Insert this new Clause to SQL/Temporal immediately before Clause 11, "Dynamic SQL".

2) Insert the following new Subclause, "Rules for period manipulation in modification statements", to SQL/Temporal at the beginning of Clause 11, "Data manipulation".

17.1 Section 11.1 Rules for period manipulation in modification statements

Function

Specify rules for period manipulation in modification statements.

General Rules

1. To remove a period RP from the valid-time period P of a row R of a table T,

   Case:

   a) If the beginning delimiting timestamp of RP is less than or equal to the beginning delimiting timestamp of P and the ending delimiting timestamp of RP is greater than the beginning delimiting timestamp of P and less than or equal to the ending delimiting timestamp of P, then replace the beginning delimiting timestamp of P for row R with the granule following the ending delimiting timestamp of RP.

   b) If the beginning delimiting timestamp of RP is less than or equal to the beginning delimiting timestamp of P and the ending delimiting timestamp of RP is greater than or equal to the ending delimiting timestamp of P, then mark row R for deletion.

   c) If the beginning delimiting timestamp of RP is greater than the beginning delimiting timestamp of P and the ending delimiting timestamp of RP is less than the ending delimiting timestamp of P, then replace the ending delimiting timestamp of P for row R with the granule preceding the beginning delimiting timestamp of RP. Let NP be a period of precision of that of P, with a beginning delimiting timestamp of the granule following the ending delimiting timestamp of RP, and an ending delimiting timestamp of that of the original value of P. A new row, with column values identical to R, and with an associated valid-time period of NP, is inserted into T.

   d) If the beginning delimiting timestamp of RP is greater than the beginning delimiting timestamp of P and less than the ending delimiting timestamp of P and the ending delimiting timestamp of RP is greater than or equal to the ending delimiting timestamp of P, then replace the beginning delimiting timestamp of P for row R with the granule preceding the beginning delimiting granule of RP.

   Note to proposal reader: If RP and P do not overlap, then do nothing.

2. To update the valid-time period P of a row R of a table T for a period UP, where PP is the precision of P,

   Case:

   a) If the beginning delimiting timestamp of UP is less than or equal to the beginning delimiting timestamp of P and the ending delimiting timestamp of UP is greater than the beginning delimiting timestamp of P and less than or equal to the ending delimiting timestamp of P, then replace the beginning delimiting timestamp of P for row R with the granule following the ending delimiting timestamp of UP. Let NP be the period of precision of PP with a beginning delimiting timestamp of that of the original value of P, and an ending delimiting timestamp of that of UP. A new row NR, with column values identical to R, and with an associated valid-time period of NP, is inserted into T. Perform the update only on NR.

b) If the beginning delimiting timestamp of UP is less than or equal to the beginning delimiting timestamp of P and the ending delimiting timestamp of UP is greater than or equal to the ending delimiting timestamp of P, then perform the update on R.

c) If the beginning delimiting timestamp of UP is greater than the beginning delimiting timestamp of P and the ending delimiting timestamp of UP is less than the ending delimiting timestamp of P, then replace the ending delimiting timestamp of P for row R with the preceding granule of the beginning delimiting timestamp of UP. Let NP be a period of precision PP, with a beginning delimiting timestamp of the granule following the ending delimiting timestamp of UP, and an ending delimiting timestamp of that of the original value of P. Two new rows, NR1 and NR2, with column values identical to R, and with an associated valid-time periods of UP and NP, respectively, are inserted into T. The update is performed only on NR1.

d) If the beginning delimiting timestamp of UP is greater than the beginning delimiting timestamp of P and less than the ending delimiting timestamp of P and the ending delimiting timestamp of UP is greater than or equal to the ending delimiting timestamp of P, then replace the ending delimiting timestamp of P for row R with the granule preceding the beginning delimiting timestamp of UP. Let NP be a period of precision PP with a beginning delimiting timestamp of that of UP, and an ending delimiting timestamp of that of the original value of P. A new row NR, with column values identical to R, and with an associated valid-time period of NP, is inserted into T. The update is performed only on NR.

Note to proposal reader: If P and UP do not overlap, then do nothing.

18 Clause 12 Data manipulation

1) Insert this new Clause to SQL/Temporal immediately following Clause 11, "'Period manipulation rules".

2) Insert the following new Subclause, "<fetch statement>", to SQL/Temporal at the beginning of Clause 12, "Date manipulation".

18.1 Subclause 12.1 <fetch statement>

Function

Position a cursor on a specified row of a table and retrieve values from that row.

Format

```
<fetch statement> ::=
        FETCH [ [ <fetch orientation> ] FROM ] <cursor name>
        [ INTO <fetch target list> ]
        [ INTO VALIDTIME <target specification> ]
```

Note to proposal reader: This extends the <fetch statement> to also allow the valid-time period of the row to be accessed.

Syntax Rules

1. (Insert this SR) <fetch statement> shall contain "INTO <fetch target list>", "INTO VALIDTIME <target specification>", or both.
2. (Insert this SR) If INTO VALIDTIME is specified in the <fetch statement>, then T shall have valid-time support. The data type of the <target specification> of <fetch statement> shall be <period type> with a precision of that of T.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) If INTO VALIDTIME is specified, then the valid-time period RP associated with the current row is assigned to the <target specification> TS, and the General Rules of Subclause 9.1, "Retrieval assignment", are applied to TS and RP as TARGET and VALUE, respectively.

Language opportunity: The FETCH statement in Dynamic SQL can be similarly extended.

18.2 Subclause 12.2 <select statement: single row>

3) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.1, "<fetch statement>".

Function

Retrieve values from a specified row of a table.

Format

```
<select statement: single row> ::=
        [ <time option> ]
        SELECT [ <set quantifier> ] <select list>
        INTO <select target list>
        <table expression>
```

Note to proposal reader: This adds an optional <time option>.

Syntax Rules

1. (Insert this SR) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in <time option>, then each exposed <table name>, <query expression>, or <correlation name> that is contained in the <table expression> without an intervening <from clause> shall identify a table with valid-time support and with identical precisions P.
2. (Insert this SR) If VALIDTIME is specified in the <valid option> that is contained in the <time option> of a <query expression> Q that is contained in the <table expression> of <select statement: single row>, then Q shall be simply contained in a <from clause>.
3. (Insert this SR) Let T be the result of <select statement: single row>.

   Case:

   a) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in <time option>, then T shall be a table with valid-time support and with precision P. The precision of <value expression> of the <valid option> that is contained in <time option> shall be P.

   b) If NONSEQUENCED VALIDTIME is specified in <time option>, then

      Case:

      i) If <value expression> is specified in the <valid option> of <time option>, then T shall be a table with valid-time support and with a precision of that of <value expression>.

      ii) Otherwise, T shall be a table without valid-time support.

   c) Otherwise, T shall be a table without valid-time support.

   Note to proposal reader: Subclause 6.1 "<item reference>" restricts the scope of column names in the <value expression> that is contained in the <valid option> that is contained in the <time option>.

   Note to proposal reader: If NONSEQUENCED is specified, then the precision of the <value expression> that is contained in the <valid option> that is contained in <time option> is arbitrary.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:

   a) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in <time option>, then the result of <table expression> TE at each valid time granule TG of precision P is the result of TE, in accordance with the General Rules of Subclause 7.7, "<table expression>", with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with its state at valid time TG. If <value expression> VE is specified in the <valid option> that is contained in <time option>, then for each row R resulting from the initial evaluation of TE,

      Case:

      i) If the value of VE and the valid-time period VP of R overlap, then the resulting valid-time period of R is the result of P_INTERSECT on the value of VE and VP.

      ii) Otherwise, R is not included in the final result of TE.

   b) If NONSEQUENCED VALIDTIME is specified in <time option>, then the the result of <table expression> TE is the result of TE, in accordance with the General Rules of Subclause 7.7, "<table expression>", with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with a table with no valid-time support with rows with identical values for the columns. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is implementation-dependent, whose data type is a <period type> with a precision of that of the valid-time period of DT, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original valid-time period of the corresponding row in DT. If <value expression> is specified in the <valid option> of <time option>, then the valid-time period of the row of the result has the value of <value expression>.

   c) Otherwise, the result of <table expression> TE is the result of TE, in accordance with the General Rules of Subclause 7.7, "<table expression>, with each of its leaf generally underlying tables with valid-time support with no intervening <from clause> replaced with its current valid-time state.

18.3 Subclause 12.3 <delete statement: positioned>

4) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.2, "<select statement: single row>".

Function

Delete a row of a table.

Format

<delete statement: positioned> ::=
    [<time option>]
    DELETE [FROM <table reference>]
        WHERE CURRENT OF <cursor name>

Note to proposal reader: This augments the production for <delete statement: positioned> with an additional, optional <time option> clause. For deletions and updates, there are four cases:

1. `DELETE`
   - works on anything
   - when applied to a table with valid-time support, deletes for times from now to forever
2. `VALIDTIME DELETE`
   - works only on tables with valid-time support
   - applies to the state at each time
3. `VALIDTIME` <period exp> `DELETE`
   - like `VALIDTIME DELETE`, but only considers the times in <period exp> (a simple example is `VALIDTIME PERIOD '[1995-01-01 - 1995-12-31]' DELETE`)
4. `NONSEQUENCED VALIDTIME DELETE`
   - works only on tables with valid-time support
   - acts like the table has an explicit timestamp column (in the <search condition>, if present)
5. `NONSEQUENCED VALIDTIME` <period exp> `DELETE`
   - like `NONSEQUENCED VALIDTIME DELETE`, but only considers the times in <period exp>

End of note.

Syntax Rules

1. (Insert this SR) Let T be the subject table of the <delete statement: positioned>.
2. (Insert this SR) If VALIDTIME is specified in <time option>, then T shall be a table with valid-time support.
3. (Insert this SR) The precision of <value expression> that is contained in the <valid option> that is contained in <time option> shall be the precision of T.

   Note to proposal reader: Subclause 6.1 "<item reference>" restricts the scope of column names in the <value expression> that is contained in the <valid option> that is contained in the <time option>.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:
   a) If VALIDTIME is specified in <time option>, then
      Case:
      i) If a <value expression> is specified in the <valid option> of <time option>, then the value of <value expression> is removed from the valid-time period of the row, in accordance with General Rule 1 of Subclause 11.1, "Rules for period manipulation in modification statements".
      ii) Otherwise, the row is marked for deletion.
   b) Otherwise,
      Case:
      i) If T is a table with valid-time support, then the period from the current timestamp to forever is removed from the valid-time period of the row, in accordance with General Rule 1 of Subclause 11.1, "Rules for period manipulation in modification statements".
      ii) Otherwise, the row is marked for deletion.

18.4 Subclause 12.4 <delete statement: searched>

5) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.3, "<delete statement: positioned>".

Function

Delete rows of a table.

Format

```
<delete statement: searched> ::=
        [ <time option> ]
        DELETE FROM <table reference>
             [ WHERE <search condition> ]
```

Note to proposal reader: This augments the production for <delete statement: searched> with an additional, optional clause.

Syntax Rules

1. (Insert this SR) Let T be the subject table of the <delete statement: searched>.
2. (Insert this SR) If VALIDTIME is specified in <time option>, then T shall be a table with valid-time support with precision P.
3. (Insert this SR) If VALIDTIME is specified in a <time option> of a <query expression> Q that is contained in the <search condition> of <delete statement: searched>, then Q shall be simply contained in a <from clause>.
4. (Insert this SR) The precision of <value expression> that is contained in the <valid option> that is contained in <time option> shall be P.

   Note to proposal reader: Subclause 6.1 "<item reference>" restricts the scope of column names in the <value expression> that is contained in the <valid option> that is contained in the <time option>.
5. (Insert this SR) If VALIDTIME is specified and NONSEQUENCED is not specified in <correlation name> that is contained in the <search condition> without an intervening <from clause> shall identify a table with valid-time support and with precision P.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:

   a) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in <time option>, then let PS be the set of those time granules TG of precision P for which the <search condition> is satisfied, in accordance with General Rule 5 of this Subclause, with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with its state at valid time TG.
   Case:

i) If <value expression> VE is specified in the <valid option> of <time option>, then for each maximally contiguous period MCP in PS, if MCP and the value of VE overlap, then the result of P_INTERSECT on MCP and the value of VE is removed from the valid-time period of the row, as well as from the valid-time period of new rows that were inserted in the processing of previous periods from PS for this row, as specified in General Rule 1 of Subclause 11.1, "Rules for period manipulation in modification statements".

ii) Otherwise, for each maximally contiguous period MCP in PS, MCP is removed from the valid-time period of the row, as well as from the valid-time period of new rows that were inserted in the processing of previous periods from PS for this row, as specified in General Rule 1 of Subclause 11.1, "Rules for period manipulation in modification statements".

b) If NONSEQUENCED VALIDTIME is specified in <time option>, then the <search condition> SC is satisfied if SC is satisfied, in accordance with General Rule 5 of this Subclause, when each leaf generally underlying table with valid-time support with no intervening <from clause> is replaced with a table with no valid-time support with rows with identical values for the columns. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is implementation-dependent, whose data type is a <period type> with a precision of that of the valid-time period of DT, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original valid-time period of the corresponding row in DT.

Case:

i) If a <value expression> is specified in the <valid option> that is contained in <time option> and the <search condition> is satisfied, then the value of <value expression> is removed from the valid-time period of the row, as specified in General Rule 1 of Subclause 11.1, "Rules for period manipulation in modification statements".

ii) Otherwise, if the <search condition> is satisfied, then the row is marked for deletion.

c) Otherwise, the <search condition> SC is satisfied if SC is satisfied, in accordance with General Rule 5 of this Subclause, when each of its leaf generally underlying tables with valid-time support with no intervening <from clause> is replaced with its current valid-time state. If the <search condition> is satisfied for the relevant row and T is a table with valid-time support, the period from the current timestamp to forever is removed from the valid-time period of the row, as specified in General Rule 1 of Subclause 11.1, "Rules for period manipulation in modification statements".

18.5 Subclause 12.5 <insert statement>

6) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.4, "<delete statement: searched>".

Function

Create new rows in a table.

Format

No additional Format items.

Syntax Rules

1. (Insert this SR) Let T be the subject table of the <insert statement>.
2. (Insert this SR) If T is a table with valid-time support with precision P, then the result of <insert columns and source> shall be a table with valid-time support of precision P.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

18.6 Subclause 12.6 <update statement: positioned>

7) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.7, "<insert statement>".

Function

Update a row of a table.

Format

```
<update statement: positioned> ::=
        [ <time option> ]
        UPDATE [ <table reference> ]
            SET <set clause list>
            WHERE CURRENT OF <cursor name>
```

Note to proposal reader: This adds an optional <time option>.

Syntax Rules

1. (Insert this SR) Let T be the subject table of the <update statement: positioned>.
2. (Insert this SR) If VALIDTIME is specified in <time option>, then T shall be a table with valid-time support with precision P.
3. (Insert this SR) The precision of <value expression> that is contained in the <valid option> that is contained in <time option> shall be P.

   Note to proposal reader: Subclause 6.1 "<item reference>" restricts the scope of column names in the <value expression> that is contained in the <valid option> that is contained in the <time option>.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:

   a) If VALIDTIME is specified in <time option>,
   Case:

      i) If <value expression> is specified, then the valid-time period of the row is updated for the value of the <value expression>, in accordance with General Rule 2 of Subclause 11.1, "Rules for period manipulation in modification statements".

      ii) Otherwise, the update is performed on the row.

   b) Otherwise,
   Case:

      i) If T is a table with valid-time support, then the valid-time period of the row is updated for the period from the current timestamp to forever, in accordance with General Rule 2 of Subclause 11.1, "Rules for period manipulation in modification statements".

      ii) Otherwise, the update is performed on the row.

18.7 Subclause 12.7 <update statement: searched>

8) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.6, "<update statement: positioned>".

Function

Update rows of a table.

Format

```
<update statement: searched> ::=
          [ <time option> ]
          UPDATE <table reference>
                SET <set clause list>
                [ WHERE <search condition> ]
```

Note to proposal reader: This adds an optional <time option>.

Syntax Rules

1. (Insert this SR) Let T be the subject table of the <update statement: searched>.
2. (Insert this SR) If VALIDTIME is specified in <time option>, then T shall be a table with valid-time support with precision P.
3. (Insert this SR) If VALIDTIME is specified in a <time option> of a <query expression> Q that is contained in the <search condition> of <update statement: searched>, then Q shall be simply contained in a <from clause>.
4. (Insert this SR) The precision of <value expression> that is contained in the <valid option> that is contained in <time option> shall be P.

   Note to proposal reader: Subclause 6.1 "<item reference>" restricts the scope of column names in the <value expression> that is contained in the <valid option> that is contained in the <time option>.
5. (Insert this SR) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in <time option>, then each exposed <table name>, <query expression>, or <correlation name> that is contained in the <search condition> without an intervening <from clause> shall identify a table with valid-time support and with precision P.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:

   a) If VALIDTIME is specified and NONSEQUENCED is not specified in the <valid option> that is contained in <time option>, then let PS be the set of those valid time granules TG of precision P for which the <search condition> is satisfied, in accordance with General Rule 5 of this Subclause, with each leaf generally underlying table with valid-time support with no intervening <from clause> replaced with its state at valid time TG.
   Case:

i) If <value expression> VE is specified in the <valid option> of <time option>, then for each maximally contiguous period MCP in PS, if MCP and the value of VE overlap, then the valid-time period of the row, as well as of new rows that were inserted in the processing of previous periods from PS for this row, are updated for the result of P_INTERSECT on MCP and the value of VE, in accordance with General Rule 2 of Subclause 11.1, "Rules for period manipulation in modification statements".

ii) Otherwise, for each maximally contiguous period MCP in PS, the valid-time period of the row, as well as of new rows that were inserted in the processing of previous periods from PS for this row, are updated for MCP in accordance with General Rule 2 of Subclause 11.1, "Rules for period manipulation in modification statements".

b) If NONSEQUENCED VALIDTIME is specified in <time option>, then the <search condition> SC is satisfied if SC is satisfied, in accordance with General Rule 5 of this Subclause, when each leaf generally underlying table with valid-time support with no intervening <from clause> is replaced with a table with no valid-time support with rows with identical values for the columns. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is implementation-dependent, whose data type is a <period type> with a precision of that of the valid-time period of DT, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original valid-time period of the corresponding row in DT.

Case:

i) If <value expression> is specified and the <search condition> is satisfied, then the valid-time period of the row is updated for the value of the <value expression>, in accordance with in General Rule 2 of Subclause 11.1, "Rules for period manipulation in modification statements".

ii) Otherwise, if the <search condition> is satisfied, then the update is performed on the row.

c) Otherwise, the <search condition> SC is satisfied if SC is satisfied, in accordance with General Rule 5 of this Subclause, when each of its leaf generally underlying tables with valid-time support with no intervening <from clause> is replaced with its current valid-time state.

Case:

i) If T is a table with valid-time support and the <search condition> is satisfied, then the valid-time period of the row is updated for the period from the current timestamp to forever, according to General Rule 2 of Subclause 11.1, "Rules for period manipulation in modification statements".

ii) Otherwise, if the <search condition> is satisfied, then the update is performed on the row.

19 Clause 12 Information Schema and Definition Schema

19.1 Subclause 12 Information Schema

1) Insert the following new Table, "TABLES view", to SQL/Temporal immediately preceding Subclause 12.2, "Definition Schema".

19.1.1 Subclause 12.1.1 TABLES view

Function

Identify the tables defined in this catalog that are accessible to a given user.

Definition

```
CREATE VIEW TABLES
         AS SELECT
               TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE,
               VALIDTIME_SUPPORT, VALIDTIME_PRECISION
         FROM DEFINITION_SCHEMA.TABLES
            WHERE ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME )
               IN (
                   SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
                    FROM DEFINITION_SCHEMA.TABLE_PRIVILEGES
                      WHERE GRANTEE IN ( 'PUBLIC', CURRENT_USER )
                   UNION
                   SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
                    FROM DEFINITION_SCHEMA.COLUMN_PRIVILEGES
                      WHERE GRANTEE IN ( 'PUBLIC', CURRENT_USER ) )
               AND TABLE_CATALOG
                = ( SELECT CATALOG_NAME FROM INFORMATION_SCHEMA_CATALOG_NAME )
```

Note to proposal reader: This adds two columns: VALIDTIME_SUPPORT and VALIDTIME_PRECISION.

Leveling Rules

No additional Leveling Rules.

19.1.2 Subclause 12.1.2 VIEWS view

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.1.1, "TABLES view".

Function

Identify the viewed tables defined in this catalog that are accessible to a given user.

Definition

```
CREATE VIEW VIEWS
              AS SELECT
                TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
                CASE WHEN ( TABLE_CATALOG, TABLE_SCHEMA, CURRENT_USER )
                            IN ( SELECT CATALOG_NAME, SCHEMA_NAME, SCHEMA_OWNER
                                        FROM DEFINITION_SCHEMA.SCHEMATA )
                     THEN VIEW_DEFINITION
                     ELSE NULL
                END AS VIEW_DEFINITION,
                CHECK_OPTION, IS_UPDATABLE,
                VALIDTIME_SUPPORT, VALIDTIME_PRECISION
              FROM DEFINITION_SCHEMA.VIEWS
                  WHERE ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME )
                    IN ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
                         FROM TABLES )
                  AND TABLE_CATALOG
                    = ( SELECT CATALOG_NAME FROM INFORMATION_SCHEMA_CATALOG_NAME )
```

Note to proposal reader: This adds two columns: VALIDTIME_SUPPORT and VALIDTIME_PRECISION.

Leveling Rules

No additional Leveling Rules.

19.1.3 Subclause 12.1.3 TABLE_CONSTRAINTS view

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.1.2, "VIEWS view".

Function

Identify the table constraints defined in this catalog that are owned by a given user.

Definition

```
CREATE VIEW TABLE_CONSTRAINTS
              AS SELECT
                  CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME,
                  TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
                  CONSTRAINT_TYPE, IS_DEFERRABLE, INITIALLY_DEFERRED,
                  VALIDTIME_SUPPORT, VALIDTIME_PERIOD
                FROM DEFINITION_SCHEMA.TABLE_CONSTRAINTS
                     JOIN
                     DEFINITION_SCHEMA.SCHEMATA S
                     ON
                      ( ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA )
                      = ( S.CATALOG_NAME, S.SCHEMA_NAME ) )
                   WHERE SCHEMA_OWNER = CURRENT_USER
                  AND CONSTRAINT_CATALOG
                    = ( SELECT CATALOG_NAME FROM INFORMATION_SCHEMA_CATALOG_NAME )
```

Note to proposal reader: This adds two columns: VALIDTIME_SUPPORT and VALIDTIME_PERIOD.

Leveling Rules

No additional Leveling Rules.

19.1.4 Subclause 12.1.4 ASSERTIONS view

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.1.3, "TABLE_CONSTRAINTS view".

Function

Identify the assertions defined in this catalog that are owned by a given user.

Definition

```
CREATE VIEW ASSERTIONS
              AS SELECT
                  CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME,
                  IS_DEFERRABLE, INITIALLY_DEFERRED,
                  VALIDTIME_SUPPORT, VALIDTIME_PERIOD
                FROM DEFINITION_SCHEMA.ASSERTIONS
                     JOIN
                     DEFINITION_SCHEMA.SCHEMATA S
                     ON
                      ( ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA )
                      = ( S.CATALOG_NAME, S.SCHEMA_NAME ) )
                   WHERE SCHEMA_OWNER = CURRENT_USER
                  AND CONSTRAINT_CATALOG
                    = ( SELECT CATALOG_NAME FROM INFORMATION_SCHEMA_CATALOG_NAME )
```

Note to proposal reader: This adds two columns: VALIDTIME_SUPPORT and VALIDTIME_PERIOD.

Leveling Rules

No additional Leveling Rules.

19.2 Subclause 12.2 Definition Schema

1) Insert the following new Table, "TABLES base table", to SQL/Temporal immediately following Subclause 12.2.1, "DATA_TYPE_DESCRIPTOR base table".

19.2.1 Subclause 12.2.2 TABLES base table

Function

The TABLES table contains one row for each table including views. It effectively contains a representation of the table descriptors.

Definition

```
CREATE TABLE TABLES
        (
        TABLE_CATALOG         INFORMATION_SCHEMA.SQL_IDENTIFIER,
        TABLE_SCHEMA          INFORMATION_SCHEMA.SQL_IDENTIFIER,
        TABLE_NAME            INFORMATION_SCHEMA.SQL_IDENTIFIER,
        TABLE_TYPE            INFORMATION_SCHEMA.CHARACTER_DATA
              CONSTRAINT TABLE_TYPE_NOT_NULL NOT NULL,
              CONSTRAINT TABLE_TYPE_CHECK CHECK ( TABLE_TYPE IN
                      ( 'BASE TABLE', 'VIEW', 'GLOBAL TEMPORARY',
                        'LOCAL TEMPORARY' ) ),
              CONSTRAINT CHECK_TABLE_IN_COLUMNS
               CHECK ( ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ) IN
                     ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
                       FROM COLUMNS ) ),
        VALIDTIME_SUPPORT     INFORMATION_SCHEMA.CHARACTER_DATA
              CONSTRAINT VALIDTIME_SUPPORT_CHECK
                CHECK (VALIDTIME_SUPPORT IN ('STATE','NONE')),
        VALIDTIME_PRECISION   INFORMATION_SCHEMA.CARDINAL_NUMBER,

        CONSTRAINT TABLES_PRIMARY_KEY
              PRIMARY KEY ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ),

        CONSTRAINT TABLES_FOREIGN_KEY_SCHEMATA
              FOREIGN KEY ( TABLE_CATALOG, TABLE_SCHEMA ) REFERENCES SCHEMATA,

        CONSTRAINT TABLES_CHECK_NOT_VIEW CHECK ( NOT EXISTS
              ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
                  FROM TABLES
                  WHERE TABLE_TYPE = 'VIEW'
                EXCEPT
                SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
                  FROM VIEWS ) )
        )
```

Note to proposal reader: This adds two columns: VALIDTIME_SUPPORT and VALIDTIME_PRECISION.

Description

1. The values of VALIDTIME_SUPPORT have the following meanings:

STATE The table being described has valid-time support.

NONE The table being described does not have valid-time support.

2. The value of VALIDTIME_PRECISION is the valid-time precision of the table being described.

19.2.2 Subclause 12.2.3 VIEWS base table

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.2.2, "TABLES base table".

Function

The VIEWS table contains one row for each row in the TABLES table with a TABLE_TYPE of 'VIEW'. Each row describes the query expression that defines a view. The table effectively contains a representation of the view descriptors.

Definition

```
CREATE TABLE VIEWS
        (
        TABLE_CATALOG      INFORMATION_SCHEMA.SQL_IDENTIFIER,
        TABLE_SCHEMA       INFORMATION_SCHEMA.SQL_IDENTIFIER,
        TABLE_NAME         INFORMATION_SCHEMA.SQL_IDENTIFIER,
        VIEW_DEFINITION    INFORMATION_SCHEMA.CHARACTER_DATA,
        CHECK_OPTION       INFORMATION_SCHEMA.CHARACTER_DATA
              CONSTRAINT CHECK_OPTION_NOT_NULL NOT NULL
              CONSTRAINT CHECK_OPTION_CHECK
                CHECK ( CHECK_OPTION IN ( 'CASCADED', 'LOCAL', 'NONE' ) ),
        IS_UPDATABLE       INFORMATION_SCHEMA.CHARACTER_DATA
              CONSTRAINT IS_UPDATABLE_NOT_NULL NOT NULL
              CONSTRAINT IS_UPDATABLE_CHECK CHECK ( IS_UPDATABLE IN ( 'YES', 'NO' ) ),
        VALIDTIME_SUPPORT  INFORMATION_SCHEMA. CHARACTER_DATA
              CONSTRAINT VALIDTIME_SUPPORT_CHECK
                CHECK (VALIDTIME_SUPPORT IN ('STATE','NONE')),
        VALIDTIME_PRECISION INFORMATION_SCHEMA.CARDINAL_NUMBER,

        CONSTRAINT VIEWS_PRIMARY_KEY
              PRIMARY KEY ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ),

        CONSTRAINT VIEWS_IN_TABLES_CHECK
              CHECK ( ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ) IN
              ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
                  FROM TABLES
                  WHERE TABLE_TYPE = 'VIEW' ) ),

        CONSTRAINT VIEWS_IS_UPDATABLE_CHECK_OPTION_CHECK
              CHECK ( ( IS_UPDATABLE, CHECK_OPTION ) NOT IN
                ( VALUES ( 'NO', 'CASCADED' ), ( 'NO', 'LOCAL' ) ) )
        )
```

Note to proposal reader: This adds two columns: VALIDTIME_SUPPORT and VALIDTIME_PRECISION.

Description

1. The values of VALIDTIME_SUPPORT have the following meanings:

   STATE The view being described has valid-time support.

   NONE The view being described does not have valid-time support.

2. The value of VALIDTIME_PRECISION is the valid-time precision of the view being described.

19.2.3 Subclause 12.2.4 TABLE_CONSTRAINTS base table

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.2.3, “VIEWS base table”.

Function

The TABLE_CONSTRAINTS table has one row for each table constraint associated with a table. It effectively contains a representation of the table constraint descriptors.

Definition

```
CREATE TABLE TABLE_CONSTRAINTS
         (
         CONSTRAINT_CATALOG      INFORMATION_SCHEMA.SQL_IDENTIFIER,
         CONSTRAINT_SCHEMA       INFORMATION_SCHEMA.SQL_IDENTIFIER,
         CONSTRAINT_NAME         INFORMATION_SCHEMA.SQL_IDENTIFIER,
         CONSTRAINT_TYPE         INFORMATION_SCHEMA.CHARACTER_DATA
         CONSTRAINT CONSTRAINT_TYPE_NOT_NULL NOT NULL
               CONSTRAINT CONSTRAINT_TYPE_CHECK
               CHECK ( CONSTRAINT_TYPE IN
                       ( 'UNIQUE',
                         'PRIMARY KEY',
                         'FOREIGN KEY',
                         'CHECK' ) ),

         TABLE_CATALOG            INFORMATION_SCHEMA.SQL_IDENTIFIER
               CONSTRAINT TABLE_CONSTRAINTS_TABLE_CATALOG_NOT_NULL NOT NULL,
         TABLE_SCHEMA             INFORMATION_SCHEMA.SQL_IDENTIFIER
               CONSTRAINT TABLE_CONSTRAINTS_TABLE_SCHEMA_NOT_NULL NOT NULL,
         TABLE_NAME               INFORMATION_SCHEMA.SQL_IDENTIFIER
               CONSTRAINT TABLE_CONSTRAINTS_TABLE_NAME_NOT_NULL NOT NULL,
         IS_DEFERRABLE            INFORMATION_SCHEMA.CHARACTER_DATA
               CONSTRAINT TABLE_CONSTRAINTS_IS_DEFERRABLE_NOT_NULL NOT NULL,
         INITIALLY_DEFERRED       INFORMATION_SCHEMA.CHARACTER_DATA
               CONSTRAINT TABLE_CONSTRAINTS_INITIALLY_DEFERRED_NOT_NULL
         VALIDTIME_SUPPORT        INFORMATION_SCHEMA.CHARACTER_DATA
               CONSTRAINT VALIDTIME_SUPPORT_CHECK
                 CHECK (VALIDTIME_SUPPORT IN ('SEQUENCED','NONSEQUENCED','NONE')),
         VALIDTIME_PERIOD         INFORMATION_SCHEMA.CARDINAL_NUMBER,

         CONSTRAINT TABLE_CONSTRAINTS_PRIMARY_KEY
               PRIMARY KEY ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME ),

         CONSTRAINT TABLE_CONSTRAINTS_DEFERRED_CHECK
               CHECK ( ( IS_DEFERRABLE, INITIALLY_DEFERRED ) IN
              ( VALUES ( 'NO',  'NO' ),
              ( 'YES', 'NO' ),
              ( 'YES', 'YES' ) ) ),

         CONSTRAINT TABLE_CONSTRAINTS_CHECK_VIEWS
               CHECK ( TABLE_CATALOG
              <> ANY ( SELECT CATALOG_NAME FROM SCHEMATA )
          OR
            ( ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ) IN
```

```
        ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
          FROM TABLES
          WHERE TABLE_TYPE <> 'VIEW' ) ) ),

CONSTRAINT TABLE_CONSTRAINTS_UNIQUE_CHECK
      CHECK ( 1 =
        ( SELECT COUNT (*)
          FROM ( SELECT CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME
                 FROM TABLE_CONSTRAINTS
                 WHERE CONSTRAINT_TYPE IN ( 'UNIQUE', 'PRIMARY KEY' )
          UNION ALL
          SELECT CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME
          FROM REFERENTIAL_CONSTRAINTS
          UNION ALL
          SELECT CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME
          FROM CHECK_CONSTRAINTS ) AS X
          WHERE ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME )
              = ( X.CONSTRAINT_CATALOG, X.CONSTRAINT_SCHEMA, X.CONSTRAINT_NAME ) ) ),

CONSTRAINT UNIQUE_TABLE_PRIMARY_KEY_CHECK
      CHECK ( UNIQUE ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
                       FROM TABLE_CONSTRAINTS
                       WHERE CONSTRAINT_TYPE = 'PRIMARY KEY' ) )
)
```

Note to proposal reader: This adds two columns: VALIDTIME_SUPPORT and VALIDTIME_PERIOD.

Description

1. The values of VALIDTIME_SUPPORT have the following meanings:

   SEQUENCED The table constraint being described was specified with VALIDTIME and without NONSEQUENCED.

   NONSEQUENCED The table constraint being described was specified with NONSEQUENCED VALIDTIME.

   NONE VALIDTIME was not specified in the table constraint being described.

2. The value of VALIDTIME_PERIOD is the value of the <value expression> contained in the <valid option> associated with the table constraint being described.

19.2.4 Subclause 12.2.5 ASSERTIONS base table

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.2.4, "TABLE_CONSTRAINTS base table".

Function

The ASSERTIONS table has one row for each assertion. It effectively contains a representation of the assertion descriptors.

Definition

```
CREATE TABLE ASSERTIONS
        (
        CONSTRAINT_CATALOG      INFORMATION_SCHEMA.SQL_IDENTIFIER,
        CONSTRAINT_SCHEMA       INFORMATION_SCHEMA.SQL_IDENTIFIER,
        CONSTRAINT_NAME         INFORMATION_SCHEMA.SQL_IDENTIFIER,
        IS_DEFERRABLE           INFORMATION_SCHEMA.CHARACTER_DATA
              CONSTRAINT ASSERTIONS_IS_DEFERRABLE_NOT_NULL NOT NULL,
        INITIALLY_DEFERRED      INFORMATION_SCHEMA.CHARACTER_DATA
              CONSTRAINT ASSERTIONS_INITIALLY_DEFERRED_NOT_NULL NOT NULL,
        CHECK_TIME              INFORMATION_SCHEMA.CHARACTER_DATA
              CONSTRAINT ASSERTIONS_CHECK_TIME_CHECK
                 CHECK ( CHECK_TIME IN ('IMMEDIATE', 'DEFERRED' ) ),
        VALIDTIME_SUPPORT       INFORMATION_SCHEMA.CHARACTER_DATA
              CONSTRAINT VALIDTIME_SUPPORT_CHECK
                  CHECK (VALIDTIME_SUPPORT IN ('SEQUENCED','NONSEQUENCED','NONE')),
        VALIDTIME_PERIOD        INFORMATION_SCHEMA.CARDINAL_NUMBER,

        CONSTRAINT ASSERTIONS_PRIMARY_KEY
              PRIMARY KEY ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME ),

        CONSTRAINT ASSERTIONS_FOREIGN_KEY_CHECK_CONSTRAINTS
              FOREIGN KEY (CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME )
                REFERENCES CHECK_CONSTRAINTS,

        CONSTRAINT ASSERTIONS_FOREIGN_KEY_SCHEMATA
              FOREIGN KEY ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA )
                REFERENCES SCHEMATA,

        CONSTRAINT ASSERTIONS_DEFERRED_CHECK
              CHECK ( ( IS_DEFERRABLE, INITIALLY_DEFERRED ) IN
                 VALUES ( ( 'NO',  'NO' ),
                          ( 'YES', 'NO' ),
                          ( 'YES', 'YES' ) ) )
         )
```

Note to proposal reader: This adds two columns: VALIDTIME_SUPPORT and VALIDTIME_PERIOD.

Description

1. The values of VALIDTIME_SUPPORT have the following meanings:

SEQUENCED The assertion being described was specified with VALIDTIME and without NONSEQUENCED.

NONSEQUENCED The assertion being described was specified with NONSEQUENCED VALIDTIME.

NONE VALIDTIME was not specified in the assertion being described.

2. The value of VALIDTIME_PERIOD is the value of the <value expression> contained in the <valid option> associated with the assertion being described.

20 Annex A (informative) Implementation-defined elements

1) Add the following Item to the list of implementation-defined elements.

   1) (Insert this Item) Subclause 7.4, "<query expression>": The name of the additional column when NONSEQUENCED VALIDTIME is specified is implementation-defined. Similarly, this occurs in Subclause 10.4, "<table constraint definition>", Subclause 10.9, "<assertion definition>", Subclause 12.2, "<select statement: single row>", Subclause 12.4, "<delete statement: searched>" and Subclause 12.7, "<update statement: searched>".

21 Acknowledgments

This change proposal was written by the four authors listed on the title page. The first author was supported in part by NSF grant ISI-9202244 and by grants from IBM, the AT&T Foundation, and DuPont. The second and third authors were supported in part by the Danish Natural Science Research Council, grant 9400911. In addition, the third author was supported by grants 11–1089–1 and 11–0061–1, also provided by the Danish Natural Science Research Council. The document was produced in part during visits by the first author to Aalborg University and by the second author to the University of Arizona.

This change proposal presents an improved and extended version of some of the constructs in TSQL2, which was designed by a committee consisting of Richard T. Snodgrass (chair), Ilsoo Ahn, Gad Ariav, Don S. Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkanri, T.Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo and Suryanarayana M. Sripada. Their participation in the TSQL2 design was critical.

We thank Curtis Dyreson for helpful comments, and Hugh Darwen and Mike Sykes for their suggested changes. Jim Melton provided extensive help on all of the proposed language extensions; the authors greatly appreciate the time Jim took to explain the intricacies of writing change proposals. Jim and Krishna Kulkarni also provided corrections to a later draft. Finally, we thank the other ANSI members for their suggestions.

A Formal Definition of Compatibility Properties

We have adopted the convention that a data model consists of two components, namely a set of data structures and a language for querying the data structures [15]. Notationally, $M = (DS,\ QL)$ then denotes a data model, M, consisting of a data structure component, DS, and a query language component, QL. Thus, DS is the set of all databases, schemas, and associated instances, expressible by M, and QL is the set of all queries in M that may be formulated on some database in DS. We use db to denote a database and q to denote a query.

A.1 Upward Compatibility

One data model is syntactically upward compatible with another data model if all data structures and legal query expressions of the latter model are contained in the former model.

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

- $\forall db_2 \in DS_2\ (db_2 \in DS_1)$ and
- $\forall q_2 \in QL_2\ (q_2 \in QL_1)$. □

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

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

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

- M_1 is syntactically upward compatible with M_2, and
- $\forall db_2 \in DS_2\ (\forall q_2 \in QL_2\ (\langle\langle q_2(db_2)\rangle\rangle_{M_2} = \langle\langle q_2(db_2)\rangle\rangle_{M_1}))$. □

This concept captures the conditions that need to be satisfied in order to allow a smooth transition from a current system, with data model M_2, to a new system, with data model M_1.

A.2 Temporal Upward Compatibility

Intuitively, the requirement is that a query q will return the same result on an associated snapshot database db as on the counterpart of the database with valid-time support, $\mathcal{T}(db)$. Further, modifications should not affect this. The precise definitions given next is explained in the following.

Definition 3: (temporal upward compatibility) Let $M_T = (DS_T, QL_T)$ and $M_S = (DS_S, QL_S)$ be temporal and snapshot data models, respectively. Also, let $\mathcal{T}$ be an operator that changes the support of a table without temporal support to the table with valid-time support with the same explicit attributes. Next, let $u_1, u_2, \ldots, u_n$ denote modification operations. With these definitions, model M_T is *temporal upward compatible* with model M_S if

- M_T is upward compatible with M_S and
- $\forall db_S \in DS_S\ (\forall q_S \in QL_S\ (\langle\langle q_S(u_n(u_{n-1}(\ldots(u_1(db_S)\ldots)))))\rangle\rangle_{M_S} =$
  $(\langle\langle q_S(u_n(u_{n-1}(\ldots(u_1(\mathcal{T}(db_S))))))\rangle\rangle_{M_T})))$. □

Assume that, when moving to the new system, some of the existing (snapshot) tables are transformed into tables with valid-time support, using `ALTER`, without changing the existing set of (explicit) attributes. This transformation is denoted by $\mathcal{T}$ in the definition. Then the same sequence of modification statements, denoted by the u_i in the definition, is applied to the snapshot and the temporal databases. Next, consider any query in the snapshot model. Such queries are also allowed in the temporal model, due to upward compatibility being required. The definition states that any such query evaluated on the resulting temporal database, using the semantics of the temporal query language, yields the same result as when evaluated on the resulting snapshot database, now using the semantics of the snapshot query language.

A.3 Sequenced Valid Semantics

We first define the notion of sequenced valid semantics among query languages. We use r and r^{vt} for denoting an instance without and with valid-time support, respectively. Similarly, db and db^{vt} are sets of table instances without and with valid-time support, respectively.

The definition uses a valid-timeslice operator $\tau_c^{M^{vt},M}$ (e.g., [9, 1]) which takes as arguments a table with valid-time support r^{vt} (in the data model M^{vt}) and a valid-time granule c and returns a table without temporal support r (in the data model M) containing all rows valid at time c. In other words, r consists of all rows of r^{vt} whose valid time includes the time granule c, but without the valid time. This operator was already introduced in Section 7.1; here, we simply have emphasized the models involved by using them as superscripts.

Definition 4: (sequenced valid semantics) Let $M = (DS, QL)$ be a snapshot relational data model, and let $M^{vt} = (DS^{vt}, QL^{vt})$ be a valid-time data model. Data model M^{vt} is *sequenced valid with respect to* data model M if

$$\forall q \in QL\,(\exists q^{vt} \in QL^{vt}\,(\forall db^{vt} \in DS^{vt}\,(\forall c\,(\tau_c^{M^{vt},M}(q^{vt}(db^{vt})) = q(\tau_c^{M^{vt},M}(db^{vt})))))). \qquad \square$$

Graphically, sequenced valid semantics implies that for all query expressions q in the snapshot model, there must exist a query q^{vt} in the temporal model so that for all db^{vt} and for all c, the commutativity diagram shown in Figure 9 holds.

$$\begin{tikzcd}
db^{vt} \arrow[r, "q^{vt}"] \arrow[d, "\text{timeslices at } c"'] & q^{vt}(db^{vt}) \arrow[d, "\text{timeslice at } c"] \\
\tau_c^{M^{vt},M}(db^{vt}) \arrow[r, "q"'] & q(\tau_c^{M^{vt},M}(db^{vt})) = \tau_c^{M^{vt},M}(q^{vt}(db^{vt}))
\end{tikzcd}$$

Figure 9: sequenced valid semantics of query q^{vt} with respect to query q at a chronon c

We require that each query q in the snapshot model has a counterpart q^{vt} in the temporal model that is sequenced valid with respect to it. Observe that q^{vt} being sequenced valid with respect to q poses no syntactical restrictions on q^{vt}. It is thus possible for q^{vt} to be quite different from q, and q^{vt} might be very involved. This is undesirable, as we would like the temporal model to be a straight-forward extension of the snapshot model. Consequently, we require that q^{vt} and q be syntactically identical.

Definition 5: (syntactically identical sequenced-valid extension) Let $M = (DS, QL)$ be a snapshot data model, and let $M^{vt} = (DS^{vt}, QL^{vt})$ be a valid-time data model. Data model M^{vt} is a *syntactically identical sequenced-valid extension* of model M if both of the following conditions hold.

1. Data model M^{vt} is sequenced valid with respect to data model M and

2. Each query in QL^{vt} that is sequenced valid with respect to a query in QL is syntactically identical to that query. □

If the valid-time data model treats tables with valid-time support as such, it is possible to use the same syntactical constructs (i.e., q^{vt} and q are identical) for querying tables with and without valid-time support. In this case, the type of support determines the meaning of the syntactical construct.

However, the identity property is incompatible with also requiring temporal upward compatibility. This latter property requires that a query from the snapshot model, when applied to a database with valid-time support, returns a table without temporal support. The property just defined requires the snapshot query to return a table with valid-time support when evaluated on the database with valid-time support.

Thus, not both of these properties can be satisfied by a temporal data model and the snapshot model it generalizes.

Our solution is to slightly relax the identity requirement, leading to the property defined below. With that property satisfied, the temporal queries may still exploit the programmers' intuition about the snapshot query language as much as possible.

Definition 6: (syntactically similar sequenced-valid extension) Let $M = (DS, QL)$ be a snapshot data model, and let $M^{vt} = (DS^{vt}, QL^{vt})$ be a valid-time data model. Data model M^{vt} is a *syntactically similar sequenced-valid extension* of model M if both of the following conditions hold.

1. Data model M^{vt} is sequenced valid with respect to data model M and
2. For each query q^{vt} in QL^{vt} that is sequenced valid with respect to a query q in QL, $q^{vt} = S_1 q S_2$, where S_1 and S_2 are text strings that depend on QL^{vt} but not on q^{vt}. □

This property is consistent with temporal upward compatibility; the language designer simply has to select at least one of S_1 or S_2 as being non-empty. For the addition to SQL/Temporal proposed here, S_1 is simply "`VALIDTIME`", and S_2 is the empty string.

A.4 Properties of SQL/Temporal

We have developed a formal denotational semantics for SQL/Temporal, in terms of the semantics of SQL3. This semantics allowed us to prove the following important properties.

- SQL/Temporal is upward compatible with SQL3.
- SQL/Temporal is temporally upward compatible with SQL3.
- The `VALIDTIME` reserved word prepended to the `SELECT` statement ensures (syntactically similar) sequenced valid semantics [1, 3].
- SQL/Temporal is temporally ungrouped [4].