ANSI X3H2-96-014
ISO/IEC JTC1/SC21/WG3 DBL ?

I S O
INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

February 21, 1996

Subject: SQL/Temporal

Status: Change Proposal

Title: Adding Valid Time — Part A

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 temporal tables 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 a smooth transition are motivated and discussed. The proposal then describes the language additions necessary to add temporal 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*. Ed. D. M. Gabbay, Ohlbach H. J.. 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*. October, 1995. (ISO/IEC JTC 1/SC 21/WG 3 DBL-LHR-002.)

[8] Melton, J. (ed.) *SQL/Temporal*. October, 1995. (ISO/IEC JTC 1/SC 21/WG 3 DBL-LHR-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 `ftp://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 (a modification to LHR-096, improving the Appendix on Proposed Language Extensions) introduces additions to SQL/Temporal to add temporal 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 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 and sequenced valid semantics, important properties that will be discussed in detail in Section 2.

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 and sequenced valid support. The next proposal will add support for non-sequenced queries and modifications. Future proposals will concern time granularities, transaction time, 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.

2 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.

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

We assume that the DBMS interface is captured in a data model and thus talk about the migration of application code using an existing data model to using a new data model. As the existing model is given, the focus is on formulating requirements to the new data model.

Much of the section is applicable to the transition from any data model to a new data model. However, we have found it convenient to assume that the transition is from a non-temporal to a temporal data model. Further, we generally assume that the transition is from the SQL3 standard to SQL/Temporal.

2.1 Upward Compatibility

Perhaps the most important aspect of ensuring a smooth transition of application code from an existing data model to a new data model 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. The next two definitions are intended to capture what is needed for that to be possible.

We adopt the convention that a data model consists of two components, namely a set of data structures and a language for querying the data structures [15]. For example, the central data structure of the relational model is the table, and the central, user-level query language is SQL.

We define a data model to be *syntactically upward compatible* with another data model if all the data structures and legal query expressions of the latter model are contained in the former model. For a data model to be *upward compatible* with another data model, we add the requirement that all queries expressible in the existing model must evaluate to the same results in both models. Syntactic upward compatibility implies that all existing databases and query expressions in the old system are also legal in the new system. The second condition guarantees that all existing queries compute the same results in the new system as in the old system. Thus, the bulk of legacy application code is not affected by the transition to a new system.

To explore the relationship between SQL/Temporal and SQL3, we employ a series of figures that demonstrate increasing query and update functionality. In Figure 1, a conventional table is 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.

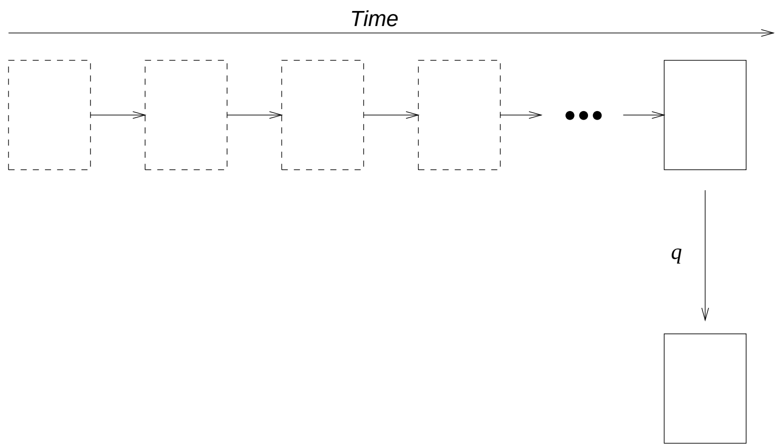

Figure 1: Level 1 evaluates an SQL3 query over a snapshot table and returns a snapshot table

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 3, 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 high salary view identifies those employees with a monthly salary greater than $3500. Then employee Therese is given a 10% raise. Since the salary table is a snapshot table, Therese's previous salary is lost. These schema changes and queries can be easily expressed in SQL3. □

2.2 Temporal Upward Compatibility

The above minimal requirements are essential to ensure a smooth transition to a new temporal data model, but they do not address all aspects of migration. Specifically, assume that an existing data model has been replaced with a new temporal model. No application code has been modified, and all tables are thus snapshot tables. Now, an existing or new application needs support for the temporal dimension of the data in one or more of the existing tables. This is best achieved by changing the snapshot table to become a temporal table (e.g., by using the `ALTER` statement of SQL/Temporal).

Note that it is undesirable to be forced to change the application code that accesses the snapshot table that is replaced by a temporal table. We formulate a requirement that states that the existing applications on snapshot tables will continue to work with no changes in functionality when the tables they access are altered to become temporal tables. 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 temporal tables.

Temporal upward compatibility is illustrated in Figure 2. When temporal 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 snapshot table.

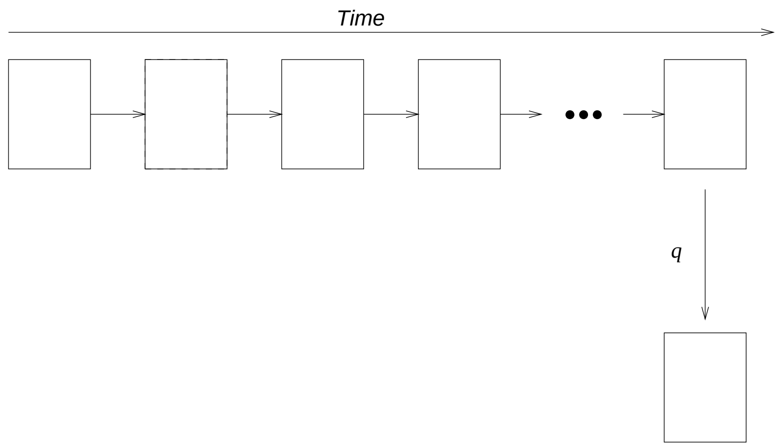

Figure 2: Level 2 evaluates an SQL3 query over a temporal table and returns a snapshot table

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 temporal table. Hence, the query only applies to the current state, and a snapshot table 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 newly-temporal tables 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 larger 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 tuple.

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

The typical situation is rather more complicated. The non-temporal database system is likely to already manage temporal data, which is encoded using snapshot tables, 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 temporal 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 temporal table: the semantics of queries and modification are retained from snapshot tables; the only difference is that intermediate states are also retained. Second, in those cases where the snapshot 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, snapshot tables that do contain temporal information and that have been converted to temporal tables can still be queried and modified by conventional SQL3 statements in a consistent manner.

2.3 Syntactically Similar Sequenced Temporal 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 by migrating to a temporal data model. 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 to a temporal model. This is achieved by exploiting the fact that programmers are likely to be comfortable with the non-temporal query language, e.g., SQL3.

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 temporal tables have semantics that are easily ("naturally") understood in terms of the semantics of the SQL3 queries on snapshot tables. 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 temporal table applies the standard SQL3 semantics on the current state of that table, resulting in a snapshot table. This figure illustrates a new query, q', which is an SQL/Temporal query. Query q' is applied to the temporal table (the sequence of states across the top of the figure), and results in a temporal table, which is the sequence of states across the bottom.

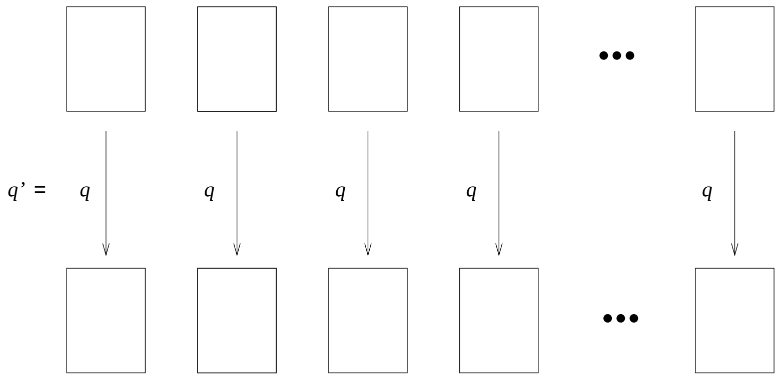

Figure 3: Level 3 evaluates an SQL/Temporal query over a temporal table and returns a temporal table

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 temporal input table, producing a state of the output table for each such application. And when q' also closely resembles q syntactically, temporal queries are easily formulated and understood. To generate query q', one needs only prepend the reserved word `VALID` to query q.

This requirement ensures that the temporal model is a user-friendly (i.e., minor) extension of the snapshot model.

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 `VALID` 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 `VALID` 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.

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 `VALID` to the `DELETE` statement, we can remove that employee from every state of the table.

Many people mispell the town Tucson as "Tuscon", perhaps because the name derives from an Indian

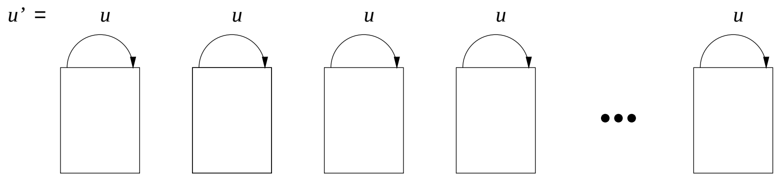

Figure 4: Level 3 also evaluates an SQL/Temporal modification on a temporal table

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 `VALID` to u. □

2.4 Non-Sequenced Queries and Modifications

In a sequenced query, the information in a particular state of the resulting temporal table 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.

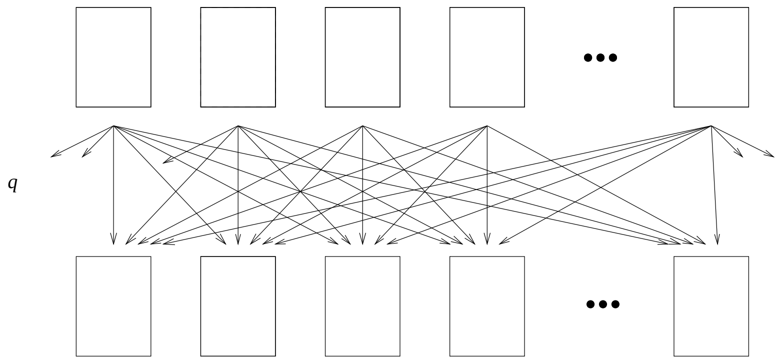

Figure 5: Level 4 evaluates a non-sequenced SQL/Temporal query over a temporal table and returns a temporal table

In this figure, two temporal tables 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 and requires additional SQL/Temporal constructs. □

As the focus of this document is on temporal upward compatibility and sequenced semantics, non-sequenced queries are accorded their own change proposal, which will be the next in the series.

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 temporal table is possibly modified, using information from possibly all states of the table before the modification. Non-sequenced modifications include future modifications.

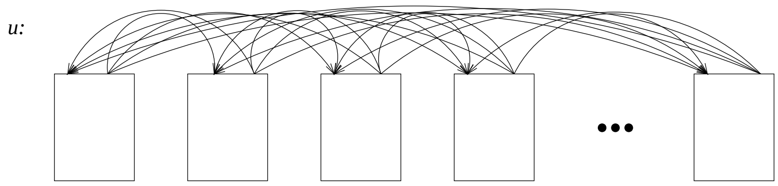

Figure 6: Level 4 also evaluates a non-sequenced SQL/Temporal modification on a temporal table

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. □

Non-sequenced modifications are clearly more complex than sequenced modifications. They require new constructs in the query language. Again, we defer discussion of these constructs and their detailed semantics to the next change proposal.

2.5 Summary

In this section, we have formulated three important requirements that SQL/Temporal should satisfy to ensure a smooth transition of legacy application code from SQL3's non-temporal data model to SQL/Temporal's data model. 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 the temporal data model be a sequenced extension of the existing data model guarantees that a core subset of the temporal data model maximally builds on the existing data model and makes the temporal query language easy to use for programmers familiar with the existing query language. The requirement thus helps protect investments in programmer training. It also turns out that this property makes the semantics of temporal tables straight-forward to specify, as shown in Section 4, and enables a wide range of implementation alternatives, some of which are listed in Section 5.2.

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 temporal query language to satisfy upward compatibility with SQL3. Thus, there is support for legacy SQL3 statements, but there are no temporal tables 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 temporal tables. 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 temporal tables. 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.

3 Supporting Temporal Tables 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 2, to support easy migration of legacy SQL3 application code [3]. The extensions concern views, assertions, and constraints (specifically temporal upward compatible and 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 "`> `".

3.1 Level 1: Upward Compatibility

Level 1 ensures upward compatibility (see Figure 1), i.e., it guarantees that legacy SQL3 statements evaluated over snapshot databases return the result dictated by SQL3.

3.1.1 SQL3 Extensions

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

3.1.2 A Quick Tour

The following statements are executed on January 1, 1995.

```
> 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)));

> 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);

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

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

> COMMIT;
```

3.2 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 valid-time database.

3.2.1 SQL3 Extensions

The create table statement is extended to define temporal tables. Specifically, this statement can be followed by the clause "`AS VALID` <datetime field>", e.g., "`AS VALID DAY`". This specifies that the table is a temporal table, with states indexed by particular days. The alter table statement is extended to permit valid-time support to be added to a snapshot table or dropped from a temporal table. The specifics may be found in Appendix F.

A temporal table is conceptually a sequence of states indexed with valid-time granules at the specified granularity. This is the view of a temporal table adopted in temporal upward compatibility and sequenced semantics.

At a more specific logical level, a temporal table is *also* a collection of rows timestamped with periods. This level is more specific because the same sequence of states may be represented by different collections of timestamped tuples. Since queries manipulate tuples with timestamps, telling what single state or sequence of states should result from a query does not *define* the query, but only restricts the possible definitions of the query. Put differently, temporal upward compatibility and sequenced semantics are required properties, not definitions.

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

3.2.2 A Quick Tour

The following statements are executed on February 1, 1995.

```
> ALTER TABLE salary ADD VALID DAY;
> ALTER TABLE employee ADD VALID 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.

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;
```

3.3 Level 3: Syntactically Similar Sequenced Language Constructs

Level 3 adds syntactically similar, sequenced, temporal counterparts of existing queries, modifications, views, constraints, and assertions (see Figure 3). Sequenced SQL/Temporal queries produce temporal tables. 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.

3.3.1 SQL3 Extensions

Temporal queries, modifications, views, assertions, and constraints are signalled by the reserved word `VALID`. This reserved word can appear in a number of locations; Appendix B supplies the details.

Derived table in a from clause In the from clause, one can prepend `VALID` to a <table subquery>.

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 `VALID`, such constraints must apply to all states of the temporal table.

Cursor expression Cursors can range over temporal tables.

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

Fetch statement The period associated with a temporal row can be placed in a local variable in embedded SQL.

Modification statements When specified with `VALID`, the modification applies to each state comprising the temporal table.

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

3.3.2 A Quick Tour

We evaluate the following statements on March 1, 1995.

Prepending `VALID` 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 `VALID`.

```
> VALID
     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.

```
> VALID
    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 - 1995-02-01

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 `VALID`.

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

Finally, we define a temporal column constraint.

```
> ALTER TABLE salary ADD VALID 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` temporal 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.

```
> VALID DELETE FROM employee
        WHERE eno = 5873;
> VALID DELETE FROM salary
        WHERE eno = 5873;
> COMMIT;
```

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

```
> VALID 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.

3.4 Level 4: Non-Sequenced Queries and Modifications

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. These constructs and their semantics will be presented in the next and future change proposals.

4 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 tuple in a temporal table. 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.

4.1 Translating SQL/Temporal Queries to Relational Algebra Expressions

We first provide the semantics of an SQL3 query over snapshot tables. In the definition given next, let $r_1, \ldots, r_n$ denote snapshot tables. We base the definition of the semantics on the semantics of SQL3, expressed in terms of the relational algebra.

$$[\![\text{<SQL–query>}]\!]_{\text{SQL/T}}(r_1, \ldots, r_n) \stackrel{\triangle}{=} [\![\text{<SQL–query>}]\!]_{\text{standard}}(r_1, \ldots, r_n)$$

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

Example 7: We start with a non-temporal query, i.e., a query evaluated with standard semantics. Assume `p` and `q` are both snapshot tables. The query Q_1

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

is equivalent to the relational algebra expression

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

□

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

$$\tau^v_c(r) \quad \stackrel{\triangle}{=} \quad \{t \mid \exists VT(\langle t \| VT \rangle \in r \wedge VT^- \leq c \wedge c < VT^+)\}$$

Example 8: We now examine a non-temporal query over a combination of snapshot and temporal tables with standard semantics. Assume `p` is a snapshot table and `t` is a temporal table. The query Q_1

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

is equivalent to the relational algebra expression

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

□

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.

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

In this definition, $[\![$ <SQL–query> $]\!]_{\mathrm{temporal}}$ is equivalent to $[\![$ <SQL–query> $]\!]_{\mathrm{standard}}$, except that every non-temporal relational algebra operator (e.g., $\bowtie, \sigma, \pi$) is replaced by a corresponding temporal relational algebra operator (e.g., $\bowtie^v, \sigma^v, \pi^v$). We provide definitions of the temporal algebra in Section 4.3.

EXAMPLE 9: An SQL/Temporal query $Q_2 = \texttt{VALID}\ Q_1$ is evaluated with temporal semantics, due to its leading valid clause. Both `p` and `q` must be temporal tables. Thus,

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

is equivalent to the temporal relational algebra expression

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

Note that apart from the ^{v}-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.

4.2 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 tuples 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 follow the data model of SQL3 and are not strictly set based.

4.3 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 tuple with valid-time [10−20] or two (value-equivalent) tuples with valid-times [10−15] and [16−20], respectively. Second, care was taken to only consider end points of period timestamps of tuples 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 "∘" denotes concatenation. The definition of AG^v is especially complex. It determines *constant periods*, during which no tuple starts or ends [10]. A constant period can go from the start of one tuple to the start of another, from the start of one tuple to the end of another, or from the end of one tuple to the end of another.

$$
\begin{array}{lcl}
\sigma^v_c(r) & \triangleq & \{\langle t \| VT\rangle \mid \langle t\|VT\rangle \in r \wedge c(\langle t\|VT\rangle)\} \\
\pi^v_f(r) & \triangleq & \{\langle t_1 \| VT\rangle \mid \langle t_2\|VT\rangle \in r \wedge t_1 = \langle f(t_2)\|VT\rangle\} \\
r_1 \cup^v r_2 & \triangleq & \{\langle t \| VT\rangle \mid \langle t\|VT\rangle \in r_1 \vee \langle t\|VT\rangle \in r_2\} \\
r_1 \bowtie^v_c r_2 & \triangleq & \{\langle\langle t_1, VT_1\rangle \circ \langle t_2, VT_2\rangle \| VT\rangle \mid \langle t_1\|VT_1\rangle \in r_1 \wedge \langle t_2\|VT_2\rangle \in r_2 \wedge \\
 & & \qquad c(\langle t_1, VT_1\rangle \circ \langle t_2, VT_2\rangle) \wedge \\
 & & \qquad VT = \mathit{intersect}(VT_1, VT_2) \wedge VT_1\ \mathit{overlaps}\ VT_2\} \\
r_1 \setminus^v r_2 & \triangleq & \{\langle t \| VT\rangle \mid \langle t\|VT_1\rangle \in r_1 \wedge \\
 & & \qquad (\exists VT_2(\langle t\|VT_2\rangle \in r_2 \wedge VT_1^- \leq VT_2^+ \wedge VT^- = VT_2^+) \vee VT^- = VT_1^-) \wedge \\
 & & \qquad (\exists VT_3(\langle t\|VT_3\rangle \in r_2 \wedge VT_1^+ \geq VT_3^- \wedge VT^+ = VT_3^-) \vee VT^+ = VT_1^+) \wedge \\
 & & \qquad VT^- < VT^+ \wedge \\
 & & \qquad \neg\exists VT_4(\langle t\|VT_4\rangle \in r_2 \ \wedge VT_4^+ > VT^- \ \wedge VT_4^- < VT^+)\} \\
AG^v_{agg,f}(r) & \triangleq & \{\langle t \circ a \| VT\rangle \mid \langle t\|VT_1\rangle \in r \ \wedge \langle t_2\|VT_2\rangle \in r \ \wedge f(t) = f(t_2) \wedge \\
 & & \qquad ((VT^- = VT_1^- \ \wedge VT^+ = VT_2^-) \vee \\
 & & \qquad\qquad (VT^- = VT_1^- \ \wedge VT^+ = VT_2^+) \vee \\
 & & \qquad\qquad (VT^- = VT_1^+ \ \wedge VT^+ = VT_2^+)) \ \wedge VT^- < VT^+ \wedge \\
 & & \qquad \neg\exists\langle t_4\|VT_4\rangle \in r(f(t) = f(t_4) \wedge \\
 & & \qquad\qquad\qquad ((VT^- < VT_4^- < VT^+) \vee (VT^- < VT_4^+ < VT^+))) \wedge \\
 & & \qquad a = agg(\{t_3 | \langle t_3\|VT_3\rangle \in r \ \wedge VT_3\ \mathit{overlaps}\ VT \wedge f(t) = f(t_3)\})\}
\end{array}
$$

Figure 7: Semantics of the temporal algebra

5 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.

5.1 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 (snapshot tables) than do temporal operators (temporal tables). 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 "v" superscript on a table indicates that it is a temporal table; those tables without such a superscript are snapshot tables, each with an explicit VT column. The auxiliary function $SN(r) = \{\langle t,\ VT\rangle \mid \langle t \| VT\rangle \in r\}$ maps a valid-time table into a snapshot table with valid-time being an explicit attribute. It is assumed that temporal tables are mapped into snapshot tables 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 well-defined over non-temporal relations only. 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 tuple variables in the calculus.

$$
\begin{array}{lll}
\sigma^v_c(r^v) & \leadsto & \sigma_c(r) \\
\pi^v_f(r^v) & \leadsto & \pi_{f,VT}(r) \\
r^v_1 \cup^v r^v_2 & \leadsto & r_1 \cup r_2 \\
r^v_1 \bowtie^v_c r^v_2 & \leadsto & \pi_{A_{r_1},VT_{r_1},A_{r_2},VT_{r_2},VT=intersect(VT_{r_1},VT_{r_2})}(r_1 \bowtie_{c \wedge VT_{r_1}\ overlaps\ VT_{r_2}} r_2) \\
r^v_1 \setminus^v r^v_2 & \leadsto & t_2 \setminus \pi_{A_{t_2},VT_{t_2}}(t_2 \bowtie_{A_{t_2}=A_{r_2} \wedge VT_{t_2}\ overlaps\ VT_{r_2}} r_2) \\
 & & t_2 = t_1 \cup \pi_{A_{r_1},period(VT^+_{r_2},VT^-_{t_1})}(t_1 \bowtie_{A_{t_1}=A_{r_2} \wedge VT^-_{t_1} \le VT^+_{r_2} \wedge VT^+_{r_2} < VT^+_{t_1}} r_2) \\
 & & t_1 = r_1 \cup \pi_{A_{r_1},period(VT^-_{r_1},VT^-_{r_2})}(r_1 \bowtie_{A_{r_1}=A_{r_2} \wedge VT^-_{r_1} < VT^-_{r_2} \wedge VT^-_{r_2} \le VT^+_{r_1}} r_2) \\
AG^v_{agg,f}(r^v) & \leadsto & AG_{agg,f}(t_2 - \pi_{A_1,A_2,a,VT^-_1,VT^+_1}(\\
 & & \qquad \sigma_{f(A_1)=f(A_2)\ \wedge((VT^-_1<VT^-_2<VT^+_1)\vee(VT^-_1<VT^+_2<VT^+_1))}(\rho_1(t_2)\times\rho_2(r)))) \\
 & & t_2 = \pi_{A_1,a,period(VT^-_1,VT^-_2)}(t_1) \cup \pi_{A_1,a,period(VT^-_1,VT^+_2)}(t_1) \cup \pi_{A_1,a,period(VT^+_1,VT^+_2)}(t_1) \\
 & & t_1 = \pi_{A_1,a,VT_1,VT_2}(\sigma_{f(A_1)=f(A_2)}(\rho_1(r)) \times \rho_2(r))
\end{array}
$$

Figure 8: Snapshot equivalents of the the temporal algebra operators

EXAMPLE 10: We continue by mapping the temporal algebraic equivalent of the SQL/Temporal query $Q_2 = \texttt{VALID}\ Q_1$ 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`.

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

□

This concludes the definition of the semantics of the proposed additions to SQL/Temporal.

5.2 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.

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 11: Continuing with the previous example, the SQL/Temporal query Q_2 on temporal tables can be mapped to an SQL3 query on snapshot tables where the implicit timestamps are now placed in explicit attributes.

```
SELECT p.X, INTERSECT(p.VT, 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` 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.

5.3 Implementing Temporal Assertions and Constraints

The general approach to checking an assertion is to negate it and then execute it as a query [3]. If the query result is empty, i.e., if no tuples are returned, the assertion is respected, otherwise it is violated.

EXAMPLE 12: To check the assertion `emp_has_sal` from Section 3.1.2 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 `VALID`, can be checked in a similar way, with a `VALID 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.

6 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 constructs proposed here support the first three levels (the last level will be addressed in detail in the next change proposal). The specific syntactic additions were outlined and examples given to illustrate these constructs. All involve simply the use of the `VALID` reserved word, to indicate temporal support (in the case of schema specification statements) and sequenced semantics (in the case of queries, modifications, views, assertions and constraints). 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. Finally, we listed alternative implementation approaches which vary in the degree of implementation difficulty and the achievable performance efficiency.

Appendix A provides formal definitions of the properties discussed in Section 2. Appendix B specifies the language additions in terms of the SQL3 language definition.

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.
- Satisfaction of temporal upward compatibility ensures that existing applications do not break when non-temporal tables are rendered temporal.
- Satisfaction of sequenced semantics ensures that temporal queries, modifications, views, assertions, and constraints are easy to specify, formalize, and implement.
- Since the semantics is defined in terms of the non-temporal semantics, the extensions are compatible with *all* the facilities of SQL3.
- 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.

7 Acknowledgements

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 paper 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 constructive criticisms and suggested changes to the proposed extensions in the Appendix.

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 temporal counterpart of the database, $\mathcal{T}(db)$. 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 type of a snapshot table to the temporal table 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 temporal tables, 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^v for denoting a snapshot and a valid-time table instance, respectively. Similarly, db and db^v are sets of snapshot and valid-time table instances, respectively.

The definition uses a valid-timeslice operator $\tau_c^{M^v,M}$ (e.g., [9, 1]) which takes as arguments a valid-time table r^v (in the data model M^v) and a valid-time granule c and returns a snapshot table r (in the data model M) containing all tuples valid at time c. In other words, r consists of all tuples of r^v whose valid time includes the time granule c, but without the valid time. This operator was already introduced in Section 4.1; 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 $M^v = (DS^v, QL^v)$ be a valid-time data model. Data model M^v is *sequenced valid with respect to* data model M if

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

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

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

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

We require that each query q in the snapshot model has a counterpart q^v in the temporal model that is sequenced valid with respect to it. Observe that q^v being sequenced valid with respect to q poses no syntactical restrictions on q^v. It is thus possible for q^v to be quite different from q, and q^v 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^v and q be syntactically identical.

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

1. Data model M^v is sequenced valid with respect to data model M and
2. Each query in QL^v that is sequenced valid with respect to a query in QL is syntactically identical to that query. $\Box$

If the valid-time data model treats valid-time tables as a new type of table, it is possible to use the same syntactical constructs (i.e., q^v and q are identical) for querying snapshot and valid-time tables. In this case, the type of a table 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 valid-time database, returns a snapshot table. The property just defined requires the snapshot query to return a temporal table when evaluated on the valid-time database.

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 $M^v = (DS^v, QL^v)$ be a valid-time data model. Data model M^v is a *syntactically similar sequenced-valid extension* of model M if both of the following conditions hold.

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

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 "`VALID`", 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 `VALID` reserved word prepended to the `SELECT` statement ensures (syntactically similar) sequenced valid semantics [1, 3].
- SQL/Temporal is temporally ungrouped [4].

B Proposal for Language Extensions

The syntax is given as extensions to "Database Language SQL — Part 2: SQL/Foundation" and "Database Language SQL — Part 7: Temporal," October, 1995 versions [7, 8]. Modifications are indicated with "•".

C Section 3.1 Definitions

Add the following terms.

g) **valid time**: The valid time of a fact is the time when the fact is true in the modeled reality.

h) **valid-time row**: A row of a table with an associated valid time, which is a value of data type PERIOD.

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

j) **snapshot table**: A snapshot table is a table with no valid-time rows.

k) **state of a valid-time table at a valid time**: The state of a valid-time table, TV, at a specified time, T, is the snapshot table comprising the rows of TV associated with valid-time periods that overlap T.

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

m) **temporal upward compatibility**: A derived table or cursor is said to be evaluated with temporal upward compatibility if its result is the same as it would be were each of its leaf generally underlying valid-time tables with no intervening <from clause> replaced with its current state.

An integrity constraint is said to be evaluated with temporal upward compatibility if its result is the same as it would be were each of its leaf generally underlying valid-time tables with no intervening <from clause> of every derived table in the integrity constraint replaced with its current state.

A delete statement, insert statement, or update statement is said to be evaluated with temporal upward compatibility if it is effectively executed for the state of the valid-time table being modified at every valid time equal to or following CURRENT_TIMESTAMP and with each of its leaf generally underlying valid-time tables with no intervening <from clause> of every derived table in the statement replaced with its current state.

n) **sequenced valid semantics**: A derived table or cursor is said to be evaluated with sequenced valid semantics if it is effectively evaluated for the state at every valid time of every leaf generally underlying valid-time table with no intervening <from clause>.

An integrity constraint is said to be evaluated with sequenced valid semantics if it is effectively evaluated for the state at every valid time of every leaf generally underlying valid-time table with no intervening <from clause>.

A delete statement, insert statement, or update statement is said to be evaluated with sequenced valid semantics if it is effectively executed at the state at every valid time both of the valid-time table being modified and of every leaf generally underlying valid-time table of every derived table with no intervening <from clause> in the statement.

D Section 5 Lexical elements

D.1 Section 5.2 <token> and <separator>

The production for the non-terminal <reserved word> is modified to add one reserved word.

<reserved word> ::=

- | `VALID`

E Section 7 Query Processing

E.1 Section 7.1 <row value constructor>

A <row value constructor element> is augmented with the ability to specify the valid-time period of a valid-time row.

<row value constructor element> ::=
  | `VALID` <period expression>

Additional syntax rules:

- At most one <row value constructor element> can contain VALID.

Additional general rules:

- If VALID is specified for a <row value constructor element>, the row is a valid-time row.

E.2 Section 7.3 <table value constructor>

Additional general rules:

1. CASE:
   - If all of the rows of the <table value constructor> are valid-time rows, then the resulting table is a valid-time table. The precision of the valid-time period of each row shall be identical, and is the precision of the valid-time of the resulting table.
   - Otherwise, no row can be a valid-time row, and the resulting table is a snapshot table.

E.3 Section 7.14 <query expression>

The production for <query expression> is augments with an optional <time option>. One of the nonterminals, <valid option>, is present to facilitate future extensions found in the TSQL2 design.

<time option> : : =
• <valid option>

<valid option> : : =
• `VALID`

<query expression> ::=
[<with clause>]
• [<time option>]
<query expression body>

Additional syntax rules:

1. If VALID is specified in <query expression>, then each exposed table, query, or correlation name contained in the <query expression body> without an intervening <from clause> shall identify a valid-time table, with identical precision of the valid-time period of the associated table.
2. VALID shall not appear in a <query expression> Q_1 that is contained in a <query expression> or SQL-data change statement except when Q_1 is simply contained in a <from clause>.

Additional general rules:

1. If VALID is not specified in <query expression>, then the <query expression body> is evaluated according to temporal upward compatibility, resulting in a snapshot table.
2. If VALID is specified in <query expression>, then the <query expression body> is evaluated according to sequenced valid semantics, results in a valid-time table with a precision of that of the exposed correlation names.

F Section 11 Schema definition and manipulation

F.1 Section 11.3 <table definition>

The production for the non-terminal <table definition> is augmented with an additional, optional clause to specify that the new table is to be a valid-time table.

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

Two productions are added.

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

<table precision> ::=
• <left paren> <period precision> <right paren>

Additional general rules:

1. If VALID is specified, the table is a valid-time table. The precision of the valid time of each row is <table precision>.
2. If VALID is not specified, the table is a snapshot table.

F.2 Section 11.6 <column definition>

An optional <time option> is added to column constraints.

<column constraint definition> ::=
[<constraint name definition>]
• [<time option>]
<column constraint> [<constraint attributes>]

Additional syntax rules:

1. If VALID is specified,

   CASE

   (a) If <column constraint> is <references specification>, then the table identified by <table name> simply contained in that construct shall be a valid-time table.

   (b) If <column constraint> is <check constraint definition>, then each table associated with an exposed table, query, or correlation name shall be a valid-time table, with identical valid-time period precisions.

Additional General Rules:

1. If no <time option> is specified, then temporal upward compatibility semantics is used.

2. A <time option> of VALID denotes sequenced valid semantics.

F.3 Section 11.10 <table constraint definition>

An optional <time option> is added to table constraints.

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

Additional syntax rules:

1. If VALID is specified in <table constraint definition>, then each exposed table, query, or correlation name contained in the <table constraint> without an interveining <from clause> shall identify a valid-time table, with identical valid-time period precisions.

Additional General Rules:

1. If no <time option> is specified, then temporal upward compatibility is used.

2. A <time option> of VALID denotes sequenced valid semantics.

F.4 Section 11.14 <alter table statement>

The <alter table statement> is augmented with the following alternatives.

<alter table action> ::=
- | <add valid definition>
- | <drop valid definition>
- | <cast valid definition>

The following productions are added.

<add valid definition> ::=
- `ADD VALID` [<table precision>]

<drop valid definition> ::=
- `DROP VALID`

<cast valid definition> ::=
- `CAST VALID AS` <table precision>

Additional syntax rules:

1. Let T be the table identified in the containing <alter table statement>.
2. For the <add valid definition>, T shall be a snapshot table.
3. For the <drop valid definition>, T shall be a valid-time table.
4. For the <cast valid definition>, T shall be a valid-time table.

Additional general rules:

1. For the <add valid definition>, T is converted from a snapshot table to a valid-time table, valid from the current timestamp to forever, with a precision of <table precision>.
2. For the <drop valid definition>, T is converted from a valid-time table to a snapshot table with contents

   `SELECT * FROM` T

   That is, the current state is retained.
3. For <cast valid definition>, the period timestamp of each tuple of T is converted to the new precision specified, via the appropriate CAST.

F.5 Section 11.43 <assertion definition>

An optional <time option> is added to assertions.

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

Additional General Rules:

1. If no <time option> is specified, then temporal upward compatibility is used.
2. A <time option> of VALID denotes sequenced valid semantics.

G Section 13 Data manipulation

G.1 Section 13.3 <fetch statement>

The <fetch statement> is extended to also allow the valid-time period of the row to be accessed.

<fetch statement> ::=
    FETCH [[<fetch orientation>] **FROM**] <cursor name>
    [**INTO** <fetch target list>]
• [**INTO VALID** <target specification>]

Additional syntax rules:

1. At least one of INTO <fetch target list> and INTO VALID <fetch target list> shall be present in a <fetch statement>.

Additional general rules:

1. The valid-time period associated with the current row is assigned to the target of the <fetch target list> following INTO VALID.

G.2 Section 13.5 <select statement: single row>

A <time option> can be prepended to this statement.

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

Additional syntax rules:

1. If VALID is specified in <time option>, then each exposed table, query, or correlation name contained in the <table expression> without an intervening <from clause> shall identify a valid-time table, with identical valid-time table precisions.

Additional general rules:

1. If VALID is not specified in <time option>, then the <table expression> is evaluated according to temporal upward compatibility, resulting in a snapshot table.
2. If VALID is specified in <time option>, then the <table expression> is evaluated according to sequenced valid semantics, resulting in a valid-time table.

G.3 Section 13.6 <delete statement: positioned>

The production for the non-terminal <delete statement: positioned> is augmented with an additional, optional clause.

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

Additional syntax rules:

1. Let T be the subject table of the <delete statement: positioned>.
2. If VALID is specified in <time option>, then T shall be a valid-time table.

Additional general rules:

1. If VALID is not specified in <time option> and T is a valid-time table, then the portion of the row's valid-time period that overlaps the period from the current timestamp to forever is removed. If the resulting valid-time period is empty, then the row itself is deleted.
2. If VALID is specified in <time option>, then the row itself is deleted.

G.4 Section 13.7 <delete statement: searched>

The production for the non-terminal <delete statement: searched> is augmented with an additional, optional clause.

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

Additional syntax rules:

1. Let T be the subject table of the <delete statement: searched>.
2. If VALID is specified in <time option>, then T shall be a valid-time table.

Additional general rules:

1. If VALID is not specified in <time option>, then the <search condition> is evaluated according to temporal upward compatibility. If the <search condition> is satisfied for the relevant row and T is a valid-time table, the portion of the row's valid-time period that overlaps the period from the current timestamp to forever is removed. If the valid-time period of the is empty, then the row itself is deleted.
2. If VALID is specified in <time option>, then the <search condition> is evaluated according to sequenced valid semantics. The valid time associated with each state satisfying the <search condition> is removed from the row's valid-time period. If the valid-time period is empty, then the row itself is deleted.

G.5 Section 13.8 <insert statement>

A <time option> may optionally be prepended to the <insert statement>.

<insert statement> ::=
- [<time option>]
  `INSERT INTO` { <table reference> | `CURSOR` <cursor name> }
  <insert columns and source>

Additional syntax rules:

1. T is the subject table of the <insert statement>.
2. Let QE be the <query expression> simply contained in <insert columns and source> If VALID is specified in <time option>, then each exposed table, query, or correlation name contained in QE without an intervening <from clause> shall identify a valid-time table with identical valid-time period precisions. Also, T shall be a valid-time table, with the same valid-time table precision.

Additional general rules:

1. If VALID is not specified in <time option>, then the <query expression> in <insert columns and source> is evaluated according to temporal upward compatibility, resulting in a snapshot table. If T is a valid-time table, the rows of the resulting snapshot table are associated with a valid-time period of the current timestamp to forever, and are inserted into T.
2. If VALID is specified in <time option>, then the <query expression> in <insert columns and source> is evaluated according to sequenced valid semantics, resulting in a valid-time table. The rows of this result are then inserted into T.

G.6 Section 13.9 <update statement: positioned>

A <time option> may optionally be prepended to the <update statement: positioned>.

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

Additional syntax rules:

1. Let T be the subject table of the <update statement: positioned>.
2. If VALID is specified in <time option>, then T shall be a valid-time table.

Additional general rules:

1. If VALID is not specified in <time option> and T is a valid-time table, then the column values associated with the portion of the row's valid-time period that overlaps the period from the current timestamp to forever are updated as specified in the <set clause list>. If the row's period of valid-time is covered by the period from the current timestamp to forever, then the entire row is updated.
2. If VALID is specified in <time option>, then the entire row is updated.

G.7 Section 13.10 <update statement: searched>

A <time option> may optionally be prepended to the <update statement: searched>.

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

Additional syntax rules:

1. Let T be the subject table of the <update statement: searched>.
2. If VALID is specified in <time option>, then T shall be a valid-time table.

Additional general rules:

1. If VALID is not specified in <time option>, then the <search condition> is evaluated according to temporal upward compatibility. If the <search condition> is satisfied for the relevant row and T is a valid-time table, the portion of the row's valid-time period that overlaps the period from the current timestamp to forever is updated as specified by the <set clause list>.
2. If VALID is specified in <time option>, then the <search condition> is evaluated according to sequenced valid semantics. For each valid time, T, for which the <search condition> is satisfied, the columns specified by the <set clause list> are updated in the state of valid time T.

H Section 18 Information Schema and Definition Schema

The base tables are extended by one column indicating whether the construct is a valid-time construct.

H.1 Section 18.3.10 TABLES base table

```
ALTER TABLE TABLES ADD COLUMN
       VALID_TIME        CHARACTER_DATA
             CONSTRAINT  VALID_TIME_CHECK
                         CHECK (VALID_TIME IN ('STATE','NONE'))
```

H.2 Section 18.3.11 VIEWS base table

```
ALTER TABLE VIEWS ADD COLUMN
       VALID_TIME        CHARACTER_DATA
             CONSTRAINT  VALID_TIME_CHECK
                         CHECK (VALID_TIME IN ('STATE','NONE'))
```

H.3 Section 18.3.15 TABLE_CONSTRAINTS base table

```
ALTER TABLE TABLE_CONSTRAINTS ADD COLUMN
       VALID_TIME        CHARACTER_DATA
             CONSTRAINT  VALID_TIME_CHECK
                         CHECK (VALID_TIME IN ('VALID','NONE'))
```

H.4 Section 18.3.16 KEY_COLUMN_USAGE base table

```
ALTER TABLE KEY_COLUMN_USAGE ADD COLUMN
       VALID_TIME        CHARACTER_DATA
             CONSTRAINT  VALID_TIME_CHECK
                         CHECK (VALID_TIME IN ('VALID','NONE'))
```

H.5 Section 18.3.17 REFERENTIAL_CONSTRAINTS base table

```
ALTER TABLE REFERENTIAL_CONSTRAINTS ADD COLUMN
       VALID_TIME        CHARACTER_DATA
             CONSTRAINT  VALID_TIME_CHECK
                         CHECK (VALID_TIME IN ('VALID','NONE'))
```

H.6 Section 18.3.18 CHECK_CONSTRAINTS base table

```
ALTER TABLE CHECK_CONSTRAINTS ADD COLUMN
       VALID_TIME        CHARACTER_DATA
             CONSTRAINT  VALID_TIME_CHECK
                         CHECK (VALID_TIME IN ('VALID','NONE'))
```

H.7 Section 18.3.22 ASSERTIONS base table

```
ALTER TABLE ASSERTIONS ADD COLUMN
       VALID_TIME        CHARACTER_DATA
             CONSTRAINT  VALID_TIME_CHECK
                         CHECK (VALID_TIME IN ('VALID','NONE'))
```