ANSI X3H2-97-010

I S O
INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

January 21, 1997

Subject: SQL/Temporal

Title: Response to MAD-220

Source: Expert's Contribution

Author: Richard T. Snodgrass

Abstract: This document responds to the points made in MAD-220 [8]. In particular, we show that the proposed constructs can be viewed as syntactic sugar, utilizing the `EXPAND` construct already in SQL/Temporal. This syntactic sugar has the advantages of allowing the user to ignore the fact that history is being maintained (temporal upward compatibility), to ask for the history of the result of a query with a single additional reserved word (sequenced), and to manipulate the table with the timestamp visible (nonsequenced).

References

[1] Böhlen, M., R.T. Snodgrass, and M.D. Soo, "Coalescing in Temporal Databases," in the *Proceedings of the International Conference on Very Large Databases*, 12 pages, September, 1996.

[2] Darwen, H. *Planned UK Contributions to SQL/Temporal*, December, 1996. (ISO/IEC JTC 1/SC 21/WG 3 DBL MAD-221).

[3] Melton, J. (ed.) *SQL/Temporal*. July, 1996. (ISO/IEC JTC 1/SC 21/WG 3 DBL-MCI-012.)

[4] Snodgrass, R. T., M. H. Böhlen, C. S. Jensen and A. Steiner. *Adding Valid Time to SQL/Temporal*. 1996. (ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2, ANSI X3H2-96-501r2.)

[5] Snodgrass, R. T., M. H. Böhlen, C. S. Jensen and A. Steiner. *Adding Transaction Time to SQL/Temporal*. 1996. (ISO/IEC JTC1/SC21/WG3 DBL MAD-147r2, ANSI X3H2-96-502r1.)

[6] Snodgrass, R.T., "Addendum to Valid- and Transaction-time Proposals," ANSI X3H2-96-582, ISO/IEC JTC1/SC21/WG3 DBL MAD-203, December, 1996.

[7] Snodgrass, R.T., "A Second Addendum to Valid- and Transaction-time Proposals," ANSI X3H2-97-010, ISO/IEC JTC1/SC21/WG3 DBL MAD-2??, January, 1997.

[8] UK response, *On Proposals for Valid-Time and Transaction-Time Support*. December 18, 1996. (ISO/IEC JTC1/SC21/WG3 DBL MAD-220)

1 Introduction

This document responds to the points made in MAD-220 [8]. Later in this document we will address each point in turn. However, continuing to call tables with temporal-time support "special tables" or "tables with hidden columns" indicates that in previous documents we have given an incorrect impression. To correct this, we will show that the proposed facilities are nothing more than carefully designed syntactic sugar, equivalent to SQL statements that use existing facilities, including the `EXPAND` operator. This is consistent with the sentiment expressed at the very end of MAD-220, which states, "If some further shorthand is really needed, it lies in **syntax**, not special kinds of tables."

2 Syntactic Sugar

The valid-time and transaction-time change proposals introduce two distinctions: tables with valid-time support and tables with transaction-time support, as well as new constructs for querying such tables. Here we argue that these enhancements do not represent a fundamental change to SQL or its implementation, by showing that the constructs can be viewed simply as syntactic sugar.

We will examine the `ALTER TABLE` statement and the `SELECT` statement, demonstrating how the new facilities can be converted into statements utilizing available constructs, include `EXPAND`. The other enhancements in the change proposal, including modifications, views, assertions, constraints, and cursors, can also be viewed in this way, as an application of the mapping described here. We emphasize that this document does not formally specify either the syntax or semantics of the temporal facilities; that information is in the change proposals. Rather, this document provides another way to view those facilities.

2.1 Data Definition

Valid-time support can be added to an existing table using `ALTER TABLE`. This is equivalent to adding a column called `VALIDTIME` to the table. Let's begin with a simple table.

```
CREATE TABLE Employee(
   Name    VARCHAR(30),
   Manager VARCHAR(30),
   Dept    VARCHAR(20))
```

The statement

```
ALTER TABLE Employee ADD VALIDTIME PERIOD(DAY)
```

is syntactic sugar for

```
ALTER TABLE Employee ADD COLUMN VALIDTIME PERIOD(DAY)
```

It also records in the schema tables that the table has valid-time support, at a granularity of `DAY`.

2.2 The Select Statement

There are three kinds of select statement: temporal upward compatible, sequenced, and nonsequenced.

2.2.1 Temporal Upward Compatibility

In a temporal upward compatible query, `VALIDTIME` is not prepended. In such a query, if a table V with valid-time support is referenced in a from clause, that reference is replaced with the following (where $C_1, \ldots, C_n$ are the columns of V before the `VALIDTIME` column was added).

`SELECT` C_1`,` $\ldots$`,` C_n `FROM` V `WHERE` V`.VALIDTIME OVERLAPS CURRENT_TIMESTAMP`

The where clause ensures that only the current state participates in the query.

As an example, the query, "which employees are not managers", can be expressed in SQL as follows.

```
SELECT Name FROM Employee WHERE Name NOT IN (SELECT Manager FROM Employee)
```

Since `Employee` has valid-time support, this query is syntactic sugar for the following.

```
SELECT Name FROM (SELECT Name, Manager, Dept
                  FROM Employee
                  WHERE VALIDTIME OVERLAPS CURRENT_TIMESTAMP) AS E
WHERE Name NOT IN (SELECT Manager
                   FROM (SELECT Name, Manager, Dept
                         FROM Employee
                         WHERE VALIDTIME OVERLAPS CURRENT_TIMESTAMP) AS E)
```

The valid-time change proposal introduces the construct `VALIDTIME(`c`)`, where c is a correlation name associated with a table with valid-time support. This is syntactic sugar for c`.VALIDTIME`.

The benefit of temporal upward compatibility is that existing non-temporal applications do not have to be changed at all when valid-time support is added to a table. The same holds true for new applications: the applications programmer can safely ignore the fact that the history is retained, if that aspect is not relevant to the application.

2.2.2 Sequenced Queries

A sequenced query is denoted by prepending `VALIDTIME`. Conceptually it evaluates the query independently on the state at each granule, yielding a resulting state valid at that granule. Such a query is syntactic sugar for the same query over the 'expanded' version of each underlying table that participates in the query. Specifically, for each table V referenced in a from clause, that reference is replaced with the following (where, again, $C_1, \ldots, C_n$ are the columns of V before the `VALIDTIME` column was added).

```
SELECT E1.C1, ..., E1.Cn, E2.EV AS VALIDTIME
FROM (SELECT C1, ..., Cn, EXPAND(VALIDTIME) AS EV FROM V) AS E1,
     TABLE(E1.EV) AS E2(EV)
```

This results in an expanded table, timestamped with a particular granule, such as a particular `DATE`. All tables over which the query is evaluated are similarly expanded.

In the query itself, if any correlation names are mentioned in the where clause, it is necessary to ensure that the value at the appropriate granule is compared. The easiest way to do this is to add the following to the where clause (here $V_1, \ldots, V_n$ are the tables mentioned in the from clause(s) of the query).

V_1`.VALIDTIME = `V_2`.VALIDTIME AND ... AND `V_{n-1}`.VALIDTIME = `V_n`.VALIDTIME`

We also need to compute the `VALIDTIME` of the result. The following should be added to the target list.

`PERIOD [ `V_1`.VALIDTIME, `V_1`.VALIDTIME ] AS VALIDTIME`

This simply constructs a period one granule in duration.

As an example, we ask, "which employees are or were not managers", can be expressed in SQL, with the proposed constructs, as follows.

```
VALIDTIME SELECT Name FROM Employee WHERE Name NOT IN (SELECT Manager FROM Employee)
```

This query is syntactic sugar for the following.

```
SELECT Name, PERIOD [ E.VALIDTIME, E.VALIDTIME ] AS VALIDTIME
FROM (SELECT E1.Name, E1.Manager, E1.Dept, E2.EV AS VALIDTIME
      FROM (SELECT Name, Manager, Dept, EXPAND(VALIDTIME) AS EV FROM Employee) AS E1,
           TABLE(E1.EV) AS E2(EV)) AS E
WHERE Name NOT IN (SELECT Manager
                   FROM (SELECT Name, Manager, Dept
                         FROM (SELECT E3.Name, E3.Manager, E3.Dept, E4.EV AS VALIDTIME
                               FROM (SELECT Name, Manager, Dept, EXPAND(VALIDTIME) AS EV
                                     FROM Employee) AS E3,
                                    TABLE(E3.EV) AS E4(EV)) AS E4
                         WHERE E4.VALIDTIME = E.VALIDTIME) AS E5)
```

Whew! What this query does is expand both mentions of `Employee` (yielding `E` and `E5`), executes the query at each time point (ensured by `E4.VALIDTIME = E.VALIDTIME`), then returns that result as being valid for a period of that one time point. Only the from clause(s) and the outermost target list are impacted. The structure of the query remains, though it is somewhat obscured by the complex subqueries in the from clauses.

The benefit is that *any* SQL query can be rendered sequenced by simply prepending `VALIDTIME`, with the mapping doing all the hard work of converting the query to use the `EXPAND` construct. A much simpler and more easily understood query syntax is thus made available.

When an aggregate is involved, we utilize the group by clause to ensure that an aggregate is evaluated separately at each point in time. As a specific case, consider the query, "Give the history of the number of employees in each department."

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

This is syntactic sugar for the following.

```
SELECT Dept, COUNT(*), PERIOD [ VALIDTIME, VALIDTIME ] AS VALIDTIME
FROM (SELECT Name, Dept, EXPAND(VALIDTIME) AS VALIDTIME FROM Employee) AS E1,
     TABLE(E1.VALIDTIME) AS E2(VALIDTIME)
GROUP BY Dept, VALIDTIME
```

As in temporal upward compatible queries, when '`*`' appears in the target list, it denotes the columns before the `VALIDTIME` column was added.

2.2.3 Nonsequenced Queries

A nonsequenced query is denoted by prepending `NONSEQUENCED VALIDTIME`. Essentially, a nonsequenced query treats the implicit time of a table with valid-time support as an explicit column. Hence, it is syntactic sugar for the same query without `NONSEQUENCED VALIDTIME`.

As an example, we ask, "which employees are or were not managers, at any time", can be expressed in SQL, with the proposed constructs, as follows. Note that any person who was an employee at one time and a manager at another time will not be included in the result.

```
NONSEQUENCED VALIDTIME SELECT Name
FROM Employee
WHERE Name NOT IN (SELECT Manager FROM Employee)
```

This query is syntactic sugar for the following.

```
SELECT Name FROM Employee WHERE Name NOT IN (SELECT Manager FROM Employee)
```

For nonsequenced queries, the `VALIDTIME` column is visible, and so is included in '`*`'.

```
NONSEQUENCED VALIDTIME SELECT * FROM Employee
```

This query is syntactic sugar for the following.

```
SELECT Name, Manager, Dept, VALIDTIME FROM Employee
```

2.3 Transaction-time Support

Transaction-time support may also be viewed as syntactic sugar, again employing the `EXPAND` construct already in SQL/Temporal. As transaction time is orthogonal to valid time, the mapping is quite similar to that for valid time.

```
ALTER TABLE Employee ADD TRANSACTION
```

This construct is syntactic sugar for the following.

```
ALTER TABLE Employee ADD COLUMN InsertTime TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP
ALTER TABLE Employee ADD COLUMN DeleteTime TIMESTAMP(3) DEFAULT NULL
```

A `DeleteTime` of `NULL` indicates the tuple has not been logically updated or deleted. As mentioned in the change proposal [5], there are significant advantages to having the system automatically maintain these columns on modifications, to ensure their integrity, which is vital for many of the applications requiring transaction-time support.

Consider first temporal upward compatibility. Here is our familiar example, "How many employees are in each department?"

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

This query is over the currently valid state of `Employee`, as best know, and is equivalent to the following.

```
SELECT Dept, COUNT(*)
FROM (SELECT Name, Dept FROM Employee
      WHERE DeleteTime IS NULL AND CURRENT_DATE OVERLAPS When) AS E1
GROUP BY Dept
```

`TRANSACTIONTIME(`c`)` is syntactic sugar for the following.

```
PERIOD [ c.InsertTime, COALESCE(c.DeleteTime, CURRENT_TIMESTAMP) ]
```

The use of NULL to indicate the row is current is exploited here in the COALESCE.

Transaction-time temporal upward compatibility works fine with all the modes of valid-time queries. As an example, to get the *history* of the number of employees, the following query suffices.

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

We retain temporal upward compatibility in transaction time (i.e., the data that has not been deleted or updated), but specify sequenced valid semantics to get the history, via VALIDTIME. The result will be a table with valid-time support. This query is syntactic sugar for the following.

```
SELECT Dept, COUNT(*), PERIOD [ VALIDTIME, VALIDTIME ] AS VALIDTIME
FROM (SELECT Name, Dept, EXPAND(VALIDTIME) AS VALIDTIME
      FROM Employee
      WHERE DeleteTime IS NULL) AS E1,
     TABLE(E1.VALIDTIME) AS E2(VALIDTIME)
GROUP BY Dept, VALIDTIME
```

Transaction-sequenced queries are handled very similarly to valid-sequenced queries. The query "When did we think that departments are overly large (over 25 employees)?" illustrates such a query, using temporal upward compatibility in valid time.

```
TRANSACTION SELECT Dept, COUNT(*)
FROM Employee
GROUP BY Dept
HAVING COUNT(*) > 25
```

This is syntactic sugar for the following query.

```
WITH E1 AS
SELECT Dept, COUNT(*), PERIOD [ TRANSACTIONTIME, TRANSACTIONTIME] AS TRANSACTIONTIME
FROM (SELECT Name, Dept, EXPAND(TRANSACTIONTIME) AS ET
      FROM (SELECT Name, Dept, PERIOD(InsertTime, COALESCE(DeleteTime, CURRENT_TIMESTAMP)
                                  AS TRANSACTIONTIME
            FROM Employee
            WHERE CURRENT_TIMESTAMP OVERLAPS VALIDTIME) AS ET) AS E1,
     TABLE(E1.ET) AS E2(TRANSACTIONTIME)
GROUP BY Dept, TRANSACTIONTIME
HAVING COUNT(*) > 25
```

All nine combinations of the three types of queries in valid time and in transaction time have analogous syntactic equivalents.

2.4 Other Statements

Temporal upward compatible modifications on tables with temporal support modify the portion from CURRENT_TIMESTAMP to the end of time. Sequenced modifications modify on a per-state basis. Nonsequenced modifications on tables with temporal support treat the implicit timestamp as an explicit column, which is what it is after the mapping.

Assertions are effectively negated queries: if no rows are returned, the assertion is respected, otherwise it is violated. Hence, temporal assertions are just syntactic sugar for conventional assertions, using the mapping described above. Constraints are syntactic sugar for assertions.

Similarly, temporal views and cursors are syntactic sugar for conventional views and cursors.

We now address each of the points made in MAD-220 [8]. We follow that document's organization, using identical section titles and numbering. Some of these comments necessitate additional changes, which are collected in a second addendum [7].

3 Temporal Upward Compatibility is Problematical

3.1 Unpleasant Consequences of Temporal Upward Compatibility

Despite the title, this section does not actually list any such consequences. In fact, the entire section 3 seems to argue that, since the advantages of temporal upward compatibility are not present *everywhere*, it is not useful *anywhere*.

3.2 Data Type ROW(...)

This argues that a new *concept* of rows is being added, and questions, where will it ever end? This line of reasoning is spurious. The change proposals add two *distinctions*, valid-time support and transaction-time support. In fact, the current SQL3 specification contains many distinctions already: "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.

Hence, the change proposals employ these two new distinctions in exactly the manner that the many other distinctions are used in the SQL3 specification. There is no need for a new row type that includes temporal support, for the same reason there are not separate row types for grouped and ungrouped tables, or for the many other distinctions already present.

3.3 Date Type TABLE(...)

All of these examples include `VALIDTIME` queries, that is, sequenced queries, and so do not really have much to do with temporal upward compatibility.

3.3.1 Variables of data type TABLE(...)

Due to temporal upward compatibility, the two `SET` statements assign the *same* value to `T`: the values of the (explicit) columns of the table. (This is a consequence of Subclauses 9.1 and 9.2 of part 2, retrieval assignment and store assignment, which is mentioned in GR2 of Subclause 13.7 <assignment statement> of PSM. In the case of the second statement, the `VALIDTIME ...` expression evaluates to a table with valid-time support. The assignment statement copies the explicit columns. The statement "the fact that two tables of different structure are assigned to the same table variable" is simply not true. Both tables have two columns, a character and an integer. Note, however, that had the `T` table included an additional `When` column, the first assignment would have been syntactically illegal, whereas it is perfectly fine if the table is altered to have temporal support.

3.3.2 Columns of data type TABLE(...)

The second insert is not allowed, because `VALIDTIME` can only be specified in the outermost <query expression> (SR 5 of Subclause 7.4 <query expression>). Tables with valid-time support *can* be nested.

3.3.3 Parameters of data type TABLE(...)

In both cases, due to the stated semantics of retrieval assignment and store assignment, only the values of explicit columns are passed.

It is certainly the case that PSM could be augmented to provide additional functionality for tables with temporal support. The absence of such

functionality does not detract from inherent advantages of temporal support. It does imply language opportunities that should be considered once temporal support is adopted.

3.4 Triggers

Those are indeed desirable additions: to provide additional support in triggers for sequenced and nonsequenced modifications (temporal upward compatible modifications are already handled fine). This is another language opportunity.

3.5 Subtables and Supertables

As noted above, the `ALTER TABLE ... ADD VALIDTIME` and `ADD TRANSACTIONTIME` can be viewed as syntactic sugar for adding one, respectively, two, columns. Hence, the same behavior concerning subtables and supertables one observes from adding a column should extend to adding temporal support. In particular, adding and dropping temporal support propagates to subtables. New GRs are necessary to specify this behavior.

3.6 The Problem of The Period Whose End Is Not Known

This does not involve temporal upward compatibility. Keeping the current state and history separate violates temporal upward compatibility. In addition, MAD-220 glosses over the problems of representing a single logical time-varying table with a pair of tables, e.g., moving rows from one to the other on changes, ensuring that they are consistent, remembering to query the appropriate one (or both), dealing with duplicates. Nevertheless, the developer is free to use separate tables. In fact, one might specify the history table as having valid-time support, thereby enabling easier expression of sequenced queries, integrity constraints, and modifications.

3.7 Insufficiency of ALTER TABLE ADD VALIDTIME

This section appears to have little to do with temporal upward compatibility. Concerning decomposing tables, that is allowed. Users are not required to avail themselves of temporal support. Concerning absorbing temporal information, this can be done with nonsequenced modifications, generally quite easily.

3.8 Problems With INSERT

The GR in question was fixed in MAD-203 [6]. Assume we have a table without temporal support. We insert a row, say indicating that employee E1 works on project P1. The table is a model of reality. This model then states that the fact is true now. If that is not indeed the case, then the model is wrong. It is this semantics that tables with valid-time support must satisfy.

The insert that is desired is simply as follows.

```
INSERT INTO WORKS_ON
NONSEQUENCED VALIDTIME PERIOD [ DATE '1997-02-02', DATE '9999-12-31' ] VALUES ('E1, 'P1')
```

This seems quite natural. If the user instead wished to use a `WORK_ON_SINCE` table, that is perfectly fine. Valid-time support is entirely optional, and is not appropriate in many cases.

3.9 Problems With DELETE

Given that the changes in reality are not entered into the database when they occurred, this is not a matter of temporal upward compatibility. Rather, it is a matter of nonsequenced updates, which are difficult in any syntax.

3.10 Problems With UPDATE

Yes, if the update occurred earlier, then the update is not a temporal upward compatibility update. There was never the claim that all modifications were temporal upward compatible. However, it *is* the case for applications that do not record the history that *their* modifications are temporal upward compatible.

Again, the authors have picked a particular case where their approach shines. Certainly there are many other cases where two tables would be onerous to manage.

4 The Specification Has Serious Deficiencies

4.1 Lack of Precise Specification

The issue raised here is an important one.

To summarize the UK's concern briefly, the change proposals do "not precisely specify the result of certain of its new kinds of queries...users can give queries without normalizing, in which case the results to be expected are not precisely specified." Clearly, the UK would be satisfied, at least on this point, if the specification were amended to require normalization. However, the real issue is not imprecision, but whether normalization should be required or optional in sequenced queries.

To be concrete, let us assume that the user declares a sequenced cursor. It is certainly true that the result of the cursor on a particular databases is not entirely specified, in that the number of rows to be returned is not specified in the change proposal. If the change proposal was altered to require normalization, then the number of rows would be known.

The current `NORMALIZE` operator eliminates duplicates as a side effect, which is undesirable. Hence, what should be used is the `NORMALIZE ALL` operator that will be proposed later [2].

There are six reasons why the valid-time and transaction-time change proposals do not specify normalization in sequenced queries. We now list these reasons for not adopting normalization for such queries.

1. *There is precedent in SQL3.*

   SQL/MM includes a text search operator whose definition does not fully specify the number of rows returned. Hence, it it seems that this property is not *a priori* grounds for disallowing a proposed construct.

2. *Normalization is arbitrary.*

   There is no canonical way to represent a sequence of states, each of which may contain duplicates, with a set of period-timestamped rows. Consider the four tables, (a) – (d), in Figure 1. For all of the rows in each table, the explicit attributes are identical; only the timestamps differ. Note that all represent the same sequence of states. Figures 1(a) and 1(b) each contain three rows, the minimum possible. Since the normalization operator hasn't yet been specified, we'll assume that it evaluates to 1(a), though it really doesn't matter which one of the two it results in. Figure 1(c), with seven rows, maximizes the periods during which nothing changes.

   Say that our `Employee` table is that shown in Figure 1(b). We now issue the following query.

   ```
   VALIDTIME SELECT * FROM Employee
   ```

   Figure 1(d) shows what would result from the sequenced queries generated by the syntactic sugar mapping given in Section 1. In terms of the syntactic sugar, there is only that one result. However, the change proposal [4] is less specific: any result that represents the same sequence of states is acceptable. So in this case, any of the four tables shown above could be returned.

   What would the user expect for the result of this query on the table shown in Figure 1(b)? If anything, the user would expect the same table back (and in fact the implementation strategy discussed in [4] does just that). However, by requiring normalization, the user is confused, because Figure 1(a) is returned. But there is no *a priori* reason why 1(a) is any better than 1(b). Hence, the semantics is picking one arbitrary representation and stating it is the *only* possible representation that can be returned. Any other representation coming in will be scrambled to yield this arbitrary result.

   So, we change the `NORMALIZE ALL` to return 1(b), instead. Now, what if the user provides a table looking like 1(a), and gives the same query. Again, the language scrambles the result, for no good reason.

   What if the original table was 1(c)? In that case, the above query takes a table containing seven rows and returns a table with three rows. Again, the language is (somewhat arbitrarily) scrambling the table.

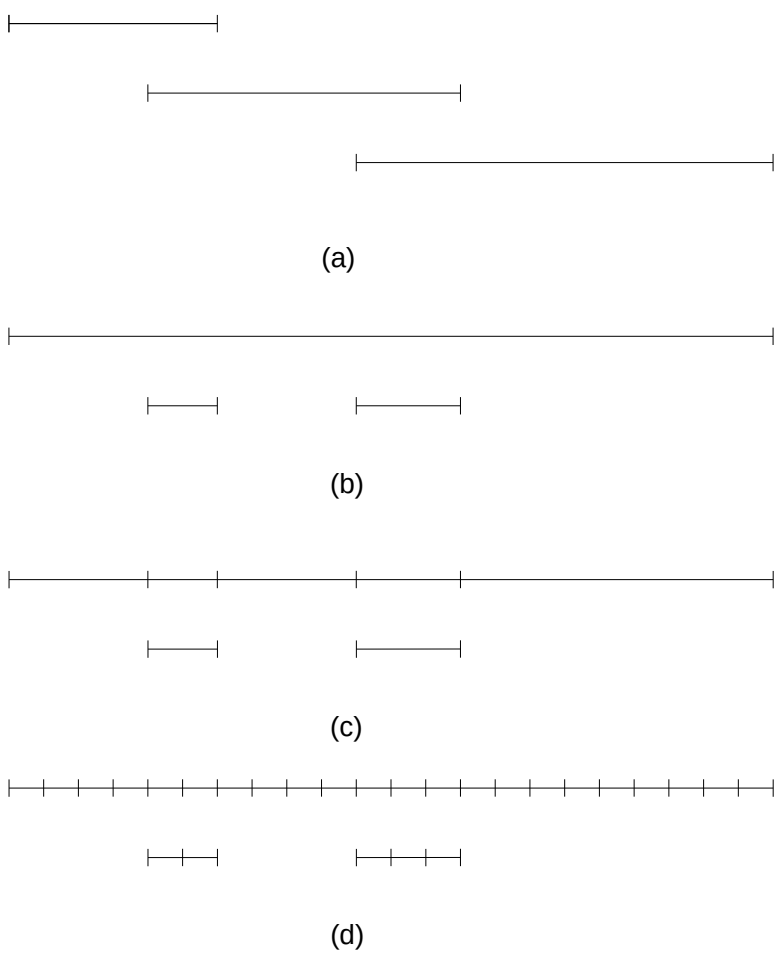

Figure 1: Different Representations of a Table With Valid-Time Support

3. *Requiring normalization does not increase expressive power.*

   If the user wanted the particular brand of normalization provided, the user can always requested it. But what if the user doesn't care? Requiring normalization does not increase the power or usability of the language. Quite the contrary, it restricts the query to *always* scramble the table.

4. *Requiring normalization constrains the implementor.*

   The most obvious way to implement the above query is simply to return the table, in whatever representation it was already in. But if normalization is required, the implementor has no choice in the matter.

5. *Requiring normalization significantly decreases the performance of all queries.*

   In the only paper thus far that focuses on the performance of normalization [1], we showed that this operation is as expensive as duplicate elimination or sorting a table. In each of these other cases, the language provides an explicit operator to request such an expensive operation. Why require an operation that is expensive that the user may not even want? Requiring normalization is counter to the design philosophy of SQL.

6. *Requiring normalization doesn't fully solve the problem.*

   Let's return to our user and her cursor. Without normalization, when she fetches the first row, she has no idea what she'll get. However, even requiring normalization, she'll still have no idea of what

she'll get. To fully specify what a cursor will return, the language would have had to mandate sorting of all queries. Currently, if the resulting table has n rows, there are $n!$ possible fully-specified tables that could result.

Our conclusion is that SQL is already non-deterministic: the result of a cursor is not fully specified. To know exactly what should be returned by a cursor, the user should both `NORMALIZE ALL` and `ORDER` the cursor. Then the user gets what she wants, and the language permits those operations not to be required if the user doesn't care (which is much of the time) and doesn't want to add the considerable expense of these operators.

In summary, normalization is similar to duplicate elimination and sorting: expensive, not always needed, and optional in the language. Had sorting been implicit in all queries, then it would have been proper to also make normalization implicit.

4.2 Row and Scalar Subqueries

The SQL specification requires in some places a <scalar subquery> and other places a <row subquery>. Both are required to evaluate to a single row. Clearly, any sequenced query that is not normalized does not fill that requirement. Examining the queries given, we can determine whether it satisfies the SRs.

1. The current state of `T` is taken. This is identical (in terms of syntactic correctness) as the nontemporal query.
2. Here `C2` is operated on as a table without temporal support, and so is identical to the nontemporal query.
3. The query will be applied to each state, and so is identical to the nontemporal query, again, in terms of syntactic correctness.
4. The same reasoning applies here.
5. Since no expression is given in `NONSEQUENCED VALIDTIME`, `T` has no temporal support, so this is illegal.
6. Here `C2` is treated as a table without temporal support, so this is fine.
7. The same holds for `T` here.
8. `T` is indeed a table without temporal support, but such tables are fine in nonsequenced queries. This is identical to the nontemporal query.

I guess I don't see the problem here.

4.3 Two Possibly Undesirable Restrictions

The issues raised here have been discussed at great length over several weeks by the authors of the change proposals (when Rick was a visiting Professor at Aalborg University in March, 1996). In fact, we have considered all the 'reasonable' queries listed here which are disallowed by the SRs of the change proposals. In the following, we give the basic rationale for these design decisions. The underlying reason in these cases generally is of the form, all the alternatives we considered exhibited more problems than the alternative eventually chosen.

4.3.1 Consequences of SR4

The first query is parsed as

```
NONSEQUENCED VALIDTIME (
        (SELECT * FROM VT)
        UNION
        (SELECT * FROM T)
        )
```

which is syntactically fine. The second query is illegal, because `VALIDTIME` does not appear in the outermost <query expression>.

The from clause is the one place where the temporal support of a query can be altered. Otherwise, the semantics of the *entire* <query expression> is selected by the prepended <time option>. This design enables the developer to put a <time option> in from of *any* <query expression>.

4.3.2 Consequences of SR3

The periodic table is a wonderful example of a table that is not time-varying. In fact, it is the only one I know of! We considered the alternative of associating with conventional tables a predefined period, such as all of time. However, there are other examples of tables that have other fixed periods of validity. It turns out that any default one picks is bound to fail in some circumstances. So a sequenced query using any one default will produce the wrong/unexpected result some of the time.

Our solution is to have the user specify that period explicitly, for each table, either within the query, as shown here, or, usually better, as a view with valid-time support. We considered it better to require the developer to state the associated period explicitly, rather than providing an implicit default that is sometimes wrong. We'll propose a language opportunity for users to specify this default on a per-table basis.

Concerning the function call, yes, the correct approach would be to prefix the query with `VALIDTIME`, and also to convert the `ELEMENT_NAME` to have valid-time support.

Suggestions for alternatives would be welcome. However, we have thought this through fairly carefully, and so are not optimistic that a better alternative can be found.

5 The Specification Has Other Non-Trivial Problems

We thank the authors for their careful reading of the change proposals. Their observations necessitate changes which appear in a second addendum [7].

6 A Possible Way Forward?

This section makes the point that "If some further shorthand is really needed, it lies in **syntax**, not special kinds of tables." As shown in Section 2, this holds for the proposed changes.

7 Summary

We sincerely appreciate the careful scrutiny of the valid-time and transaction-time proposals carried out by the UK. It is only by such means that the best possible design for such support will result. Addressing the concerns of MAD-220 necessitated over 30 separate changes, all improving the change proposal.

There has been a great deal of concern voiced over the necessity of special types of tables. We apologize if the impression had been conveyed that fundamental changes to the data model were being proposed. We hope that the demonstration that the facilities are merely convenient syntactic sugar will ameliorate this concern. Perhaps if we had thought to present it this way before, faster progress could have occurred.

Some of the problems concerning temporal upward compatibility seem to arise from misunderstandings of the specification, some were addressed in the addendum, some are inherent in all existing proposals for temporal support (including the facilities now in SQL/Temporal), some are more properly seen as language opportunities, to be addressed with additional constructs, and the rest are addressed in a subsequent addendum.

It appears that a basis for the concern about temporal upward compatibility (TUC) is that the perceived claim was that it (TUC) applied to all non-temporal applications, and with slight changes, to all applications. We are sorry if we conveyed that impression. We agree that non-temporal applications are in the minority. We also agree that many temporal applications utilize temporal columns in ways that make it difficult to convert such tables to valid-time support. Finally, we agree that there are applications that could benefit from other organizations, including the two-table design discussed in MAD-220.

That said, the fact that temporal upward compatibility is syntactic sugar, that it is completely optional, and that it is enthusiastically embraced by users whenever they hear of it, indicates that it would be useful to support in SQL.

Concerning the issue of specification precision, we showed that the issue comes down to making normalization optional or required for sequenced queries. Our rationale for optional normalization follows closely that for optional sorting, and for optional duplicate elimination.