A n O v e r v i e w a n d A n a l y s i s

o f P r o p o s a l s

B a s e d o n t h e T S Q L 2 A p p r o a c h

by

Hugh Darwen and **C. J. Date**

*Date of this **DRAFT:** March 10th, 2005*

1. INTRODUCTION

Along with Nikos Lorentzos, the present writers have described in detail, in reference [6], an approach to the temporal database problem that is firmly rooted in the relational model of data (and we assume here and there in the present paper that you have some familiarity with the ideas of reference [6]). However, many other approaches have been proposed and described in the literature. In this paper, we take a brief look at the "temporal query language" **TSQL2,** which is probably the best known and most influential of those alternative approaches——indeed, a version of it was even proposed at one time for inclusion in the SQL standard (see Section 2 below).[1] Sections 3-7 provide an overview of the major features of TSQL2. Sections 8-11 then describe what we regard as a series of major flaws in the TSQL2 approach, and Section 12 offers a few concluding remarks.

With regard to those "major flaws," incidentally, we should say there is one that seems to us so significant——indeed, it underlies all the rest——that it needs to be mentioned right away,

[1] Reference [1] describes a temporal query language very similar to TSQL2 called ATSQL. In this paper, we use the name *TSQL2* as a convenient generic label to refer to the approach espoused in all or any of references [1] and [11-14].

and that is that **TSQL2 involves "hidden attributes."**[2] As a direct consequence of this fact, the basic data object in TSQL2 is not a relation, and the approach thus clearly violates *The Information Principle*. In other words, TSQL2, whatever else it might be, is certainly not relational. We should immediately add that TSQL2 is not alone in this regard——most of the other temporal proposals described in the literature do the same thing, in one way or another. What is more, the picture is muddied by the fact that most if not all of the researchers involved refer to their proposals, quite explicitly, as *relational* approaches to the problem, even though they are clearly not (relational, that is).

We will elaborate on this matter of hidden attributes in Sections 3 and 9. And although our remarks in those sections are framed in terms of TSQL2 specifically, it should be clear that those remarks apply with equal force, *mutatis mutandis,* to *any* approach that attempts to "hide attributes" in the same kind of way that TSQL2 does.

One final preliminary remark: Our discussions of TSQL2——which are not meant to be exhaustive, please note——are based primarily on our own understanding of references [13-15]. Naturally we have tried to make those discussions as accurate as we can, but it is of course possible that we have misinterpreted those references on occasion. If so, we apologize; in our defense, however, we need to say that those references [13-15] do contradict one another on occasion.

2. TSQL2 AND THE SQL STANDARD

First of all, a little background. The body that publishes the international SQL standard is the International Organization for Standardization ("ISO"). That body produced versions of the standard in 1992 (*SQL:1992,* known informally as SQL2) and 1999 (*SQL:1999,* known informally as SQL3). SQL:1999 [8] is the version of the standard that is current at the time of writing; a thorough tutorial description of the previous version, SQL:1992, with an appendix giving an overview of SQL3 as it was around 1996 or so, can be found in reference [5]. The next version is likely to be ratified later this year (2003). *Note added later:* In fact this latter did happen, and SQL:2003 is now the current standard.

The ISO committee with direct responsibility for the SQL standard has delegates representing a variety of national standards bodies. During the 1990s, the United States national body received a proposal for a set of temporal extensions to SQL based on TSQL2. (The name "TSQL2" presumably reflects the fact that the language was designed as an extension to SQL2 specifically [11], which——in the form of SQL:1992——was the official standard at the time.) The US national body in turn

[2] The TSQL2 term is *implicit columns*. Regular attributes are called *explicit* columns.

submitted that proposal as an "individual expert's contribution" (i.e., not as a formal position paper) for consideration by the ISO SQL committee [13].

The ISO SQL committee proceeded to examine the proposal carefully. As part of that examination, members of the United Kingdom national body in particular came to the conclusion that, while the proposal might look attractive at first glance, that attractiveness did not stand up to close scrutiny. To be specific, they found that TSQL2 departed significantly from both established language design principles in general [2] and relational database theory in particular (as already noted). What is more, they found that the departures in question were significantly different in kind from SQL's other well-documented departures from those principles and that theory. As a consequence of those findings, the UK body prepared a paper [4] and submitted it for consideration at the ISO committee meeting in January 1997.

The UK paper demonstrated conclusively that the specific proposals of reference [13] were unacceptable for the working draft of SQL3 at that time. Indeed, it went further: It showed why the UK body was unlikely ever to support any proposal that was based on TSQL2. Actually, the UK opposition to such an approach had become clear to other participants at previous ISO committee meetings in 1995 and 1996. However, those previous meetings had at least achieved the following positive results among others:

- Agreement had been reached that temporal extensions of some kind were desirable.
- A working draft for a possible Part 7 of the international standard, known informally as "SQL/Temporal," had been established as the base document for such extensions.
- Finally, discussion papers suggesting various ways forward had been considered and debated.

Moreover, despite the arguments of reference [4], several members of the ISO committee remained enthusiastic about the possibility of a TSQL2-based approach——perhaps because of the apparent reluctance on the part of the TSQL2 proponents themselves to acknowledge that the arguments of reference [4] held water. Be that as it may, it was at least agreed that the specific proposals of reference [13] needed a significant amount of revision and left several important questions unanswered, and the US delegates therefore agreed to withdraw the submission.[3] It was further

[3] In spite of that withdrawal (and in support of our claim above that the TSQL2 proponents themselves seem reluctant to accept the arguments of reference [4]), we observe that (a) Chapter 12 of reference [15], published some three years later, continues to describe the TSQL2-based proposals as if they were part of SQL3, and (b) reference [1], published later still, continues to pursue the idea of *statement modifiers,* even though statement modifiers were one of the

agreed that nobody would submit any more temporal proposals to the ISO committee until SQL3 was formally published. That publication took place at the end of 1999, when the informal name SQL3 was replaced by the official one, SQL:1999.

Following all of the activity described above, ISO interest in temporal extensions waned somewhat; in fact, nobody was prepared to spend any more time on the matter unless and until some positive move was made by the leading SQL vendors. And time ran out toward the end of 2001, when——since no vendor had made any such move, and the committee had therefore done no further work on the project to develop Part 7 of the standard——ISO's own bylaws led to that project being canceled altogether. At the time of writing, therefore, the working draft document mentioned above ("Part 7: SQL/Temporal") is in limbo.

3. TSQL2 TABLES

We begin our description of TSQL2 by describing the basic data objects——to be more specific, the various kinds of "tables"——that TSQL2 supports ("tables" in quotes because those "tables" are certainly not relational tables, as we will quickly see). Then, in subsequent sections, we can go on to explain various detailed aspects of TSQL2 in terms of those different kinds of tables.

Before we can even start to describe those tables, however, we need to say a word about terminology. As previously stated, TSQL2 is designed as an extension to SQL specifically. As a result, we will frequently be forced to use SQL terminology instead of our own preferred terms (as documented in reference [6]) in our explanations in this paper. However, we will do our best to stay with our preferred terms as much as possible——certainly when we are talking about general concepts rather than TSQL2 specifics.

Here then is a list of our preferred terms and their SQL or TSQL2 counterparts. Note that we do not say "equivalents," because the SQL (or TSQL2) terms are mostly not equivalent to their relational counterparts. For example, a tuple and a row are not the same thing, nor are an attribute and a column.

TSQL2 ideas that reference [4] showed to be fundamentally flawed (see Section 8).

Our term	SQL or TSQL2 term
relvar	table
relation	table
tuple	row
attribute	column
interval	period
stated time	valid time
logged time	transaction time
operator	operator, function

Now to the question of the kinds of tables that TSQL2 supports. Consider Fig. 1, which shows a sample value for a relvar called S_DURING_LOG, with attributes S# (supplier number), DURING (stated or "valid" time), and X_DURING (logged or "transaction" time). Note our use of symbols of the form *d01, d02,* etc., in that figure; the "*d*" in those symbols can conveniently be pronounced "day," a convention to which we will adhere throughout this paper. We assume that day 1 immediately precedes day 2, day 2 immediately precedes day 3, and so on; also, we drop insignificant leading zeros from expressions such as "day 1" (as you can see). *Note:* Details of how relvar S_DURING_LOG is meant to be interpreted can be found in reference [6]; here we just note that, for the sake of the example, we have assumed that day 99 is "the end of time." We have also assumed that today is day 10 (and we will stay with that assumption throughout the rest of this paper).

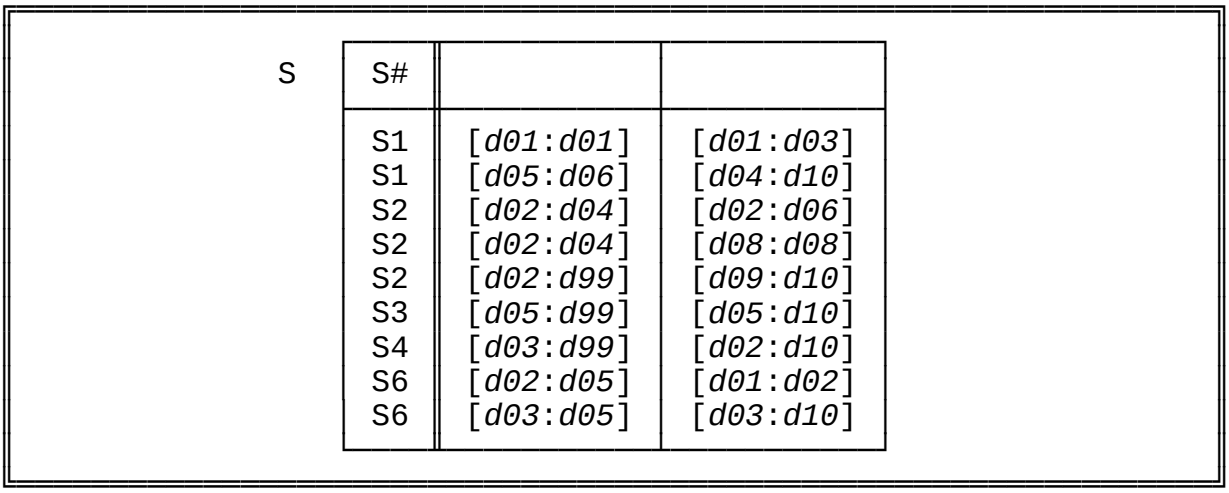
S_DURING_LOG

S#	DURING	X_DURING
S1	[*d01*:*d01*]	[*d01*:*d03*]
S1	[*d05*:*d06*]	[*d04*:*d10*]
S2	[*d02*:*d04*]	[*d02*:*d06*]
S2	[*d02*:*d04*]	[*d08*:*d08*]
S2	[*d02*:*d99*]	[*d09*:*d10*]
S3	[*d05*:*d99*]	[*d05*:*d10*]
S4	[*d03*:*d99*]	[*d02*:*d10*]
S6	[*d02*:*d05*]	[*d01*:*d02*]
S6	[*d03*:*d05*]	[*d03*:*d10*]

Fig. 1: Relvar S_DURING_LOG——sample values

Fig. 2 shows a table that might be regarded as a TSQL2 counterpart to the relvar shown in Fig. 1. Note the following points right away:

- The table is named S, not S_DURING_LOG.
- The table has no double underlining to indicate a primary key.
- The "timestamp" columns——i.e., the columns corresponding to attributes DURING and X_DURING——are unnamed.
- Those timestamp columns are separated from the rest of the table by a double vertical line.



S

S#		
S1	[*d01*:*d01*]	[*d01*:*d03*]
S1	[*d05*:*d06*]	[*d04*:*d10*]
S2	[*d02*:*d04*]	[*d02*:*d06*]
S2	[*d02*:*d04*]	[*d08*:*d08*]
S2	[*d02*:*d99*]	[*d09*:*d10*]
S3	[*d05*:*d99*]	[*d05*:*d10*]
S4	[*d03*:*d99*]	[*d02*:*d10*]
S6	[*d02*:*d05*]	[*d01*:*d02*]
S6	[*d03*:*d05*]	[*d03*:*d10*]

Fig. 2: A TSQL2 bitemporal table

The object depicted in Fig. 2 is an example of what TSQL2 calls a *bitemporal table*. Let us examine it more carefully. First of all, the unnamed timestamp columns are *hidden from the user,* which is why we show them separated from the rest of the table by that double vertical line. (To the user, in other words, the table contains just one column, named S#.) Of course, there has to be a way to access those hidden columns, and so there is, as we will see near the end of Section 6; however, that access cannot be done in regular relational fashion——i.e., by simply referring to the columns by name——because, to repeat, they have no names. Indeed, those hidden columns are not relational attributes, and the overall table is not a relation (more precisely, it is not a relvar).

Next, the table is named S, not S_DURING_LOG, because TSQL2 wants to pretend as far as possible that the table is indeed just the usual suppliers table;i[4] to say it again, the timestamp columns are hidden. In particular, TSQL2 wants regular SQL statements to operate on the table, so far as the user is concerned, just as if those hidden columns were not there. (Indeed, it wants much more than that, as we will see in Section 6.)

[4] We are assuming here a version of "the usual suppliers table" that has just one column, called S# ("supplier number").

Next, we have omitted the double underlining we normally use to indicate a primary key, because we clearly cannot pretend to the user that the combination of all three columns is the primary key (as it really is, in effect), while at the same time pretending to that same user that the hidden columns are not there. (In fact, TSQL2 also wants to pretend, in effect, that certain rows are not there either, as we will also see in Section 6; as a consequence of this latter pretense, it is able to pretend as well that {S#} alone is the primary key. But this notion is hard to illustrate in a figure like Fig. 2, and we have not attempted to do so.)

Now we need to explain that both of the hidden columns are in fact optional, in general. As a result, TSQL2 supports at least four kinds of tables:

- A **bitemporal table** is one that includes exactly two hidden columns, one containing "valid-time" timestamps and the other "transaction-time" timestamps.
- A **valid-time state table**[5] (or just *valid-time table* for short) is one that includes exactly one hidden column, which contains "valid-time" timestamps.
- A **transaction-time state table** (or just *transaction-time table* for short) is one that includes exactly one hidden column, which contains "transaction-time" timestamps.
- A **regular table** (note that we cannot say "just *table* for short," because *table* is a generic term that now has to encompass all of the new kinds of tables introduced by TSQL2 as well as regular tables *per se*) is a table that includes no hidden columns at all.

More terminology: A table with a valid-time hidden column is said to be *a table with valid-time support*. A table with a transaction-time hidden column is said to be *a table with transaction-time support*. A table with either valid-time support or transaction-time support is said to be *a table with temporal support*.

Finally——this is important!——note that in TSQL2 valid- and transaction-time columns are always **hidden by definition**. A user-visible column that happens to contain valid or transaction times is not regarded by TSQL2 as a valid- or transaction-time column at all, but rather as a column that contains what it calls *user-defined times*. From this point forward, therefore, we will assume

[5] The term *state* here corresponds to reference [6]'s use of the term **during**——i.e., it refers to the idea that something is true, or believed to be true, throughout some period (interval). It is contrasted with the term *event,* which corresponds to reference [6]'s use of the term **at**——i.e., it refers to the idea that something is true (or believed to be true) at a certain point in time.

that all valid-time columns and all transaction-time columns are hidden, barring explicit statements to the contrary (though we will often refer to such columns explicitly as "hidden columns," for clarity). See the annotation to reference [10] for further discussion.

4. THE CENTRAL IDEA

We now proceed to describe what we perceive to be the central idea of TSQL2. Now, as every student of temporal databases quickly becomes aware, queries involving intervals (temporal or otherwise) can be surprisingly tedious or difficult or both to express. And while it is true that the various operators discussed in reference [6]——Allen's operators, PACK and UNPACK, and (especially) the so-called "U_ operators"——can help in this regard, some degree of both tedium and difficulty still remains, even when those shorthands are used. Accordingly, it is a goal of TSQL2 to simplify matters still further. And it appears that such further simplification might be possible in a certain very special case; to be specific, it might be possible if and only if the query under consideration satisfies all four of the following conditions (labeled *C1*-*C4* for purposes of subsequent reference).

C1: The output table——i.e., the final result——has at most one (hidden) valid-time column and at most one (hidden) transaction-time column.

C2: The output table has at least one additional (regular) column, over and above any hidden valid- or transaction-time column.

C3: Every input or intermediate-result table satisfies these same properties——at most one hidden valid-time column, at most one hidden transaction-time column, and at least one additional regular column.

C4: Every hidden valid-time column in every input, output, or intermediate-result table involved at any point in the query is of exactly the same data type.[6]

Let us examine these conditions a little more carefully. Here again is the first one (now stated a little more simply):

C1: The result has at most one valid-time column and at most one transaction-time column.

This condition clearly derives from the fact that TSQL2 tables have at most one valid-time column and at most one transaction-time column (both hidden, of course). Here by way of example is a **Tutorial D** query that satisfies the condition (though of course the "valid- and transaction-time columns"——i.e., the stated- and

[6] In fact Condition *C4* applies to transaction-time columns as well, but transaction times in TSQL2 are always of a data type that is chosen by the DBMS.

logged-time attributes, in the terminology of reference [6]——are not hidden in **Tutorial D**):

```
WITH ( SP_DURING RENAME ( P# AS XP# ) ) AS T1 ,
     ( SP_DURING RENAME ( P# AS YP# ) ) AS T2 :
USING DURING * T1 JOIN T2 *
```

Note: Relvar SP_DURING represents the predicate "Supplier S# was able to supply part P# throughout interval DURING." Thus, the query returns a result with predicate "Supplier S# was able to supply both part XP# and part YP# throughout interval DURING." That result thus certainly does have at most one stated-time attribute and at most one logged-time attribute; in fact, it has exactly one stated-time attribute, called DURING (which shows when supplier S# was able to supply both part XP# and part YP#), and no logged-time attribute at all.

As a matter of fact, this same query satisfies Conditions *C3* and *C4* as well. Here again are those conditions (now slightly simplified):

C3: Every input or intermediate-result table has at most one valid-time column, at most one transaction-time column, and at least one additional column.

C4: Every valid-time column in every input, output, or intermediate-result table is of exactly the same data type.

Condition *C3* derives from two facts: first, the fact that, again, TSQL2 tables have at most one (hidden) valid-time column and at most one (hidden) transaction-time column; second, the fact that regular SQL tables must have at least one column. Condition *C4* derives, in part, from the fact that TSQL2 makes use of *statement modifiers* to express queries (as we will see in Section 6), and those modifiers are "global," in the sense that they are meant to apply uniformly to every table involved in the query in question. (We say "meant to" here advisedly; whether they actually do so is another matter. See Section 8.)

Anyway, to revert to the **Tutorial D** example:

- (Condition *C3*) The relations denoted by SP_DURING, T1, and T2 each have exactly one stated-time attribute (called DURING in every case), as does the final result relation. In the case of SP_DURING, for example, attribute DURING shows when supplier S# was able to supply part P#; in the case of the final result, it shows when supplier S# was able to supply both part XP# and part YP#. Furthermore, the relations denoted by SP_DURING, T1, and T2 each have at least one additional attribute and no logged-time attribute at all.
- (Condition *C4*) Attribute DURING is clearly of the same type in every case: namely, type INTERVAL_DATE.

Back now to Condition *C1*. Here by contrast is a **Tutorial D** query that does *not* satisfy that condition, *mutatis mutandis:*

```
WITH ( SP_DURING RENAME ( P# AS XP#,
                          DURING AS XDURING ) ) AS T1 ,
     ( SP_DURING RENAME ( P# AS YP#,
                          DURING AS YDURING ) ) AS T2 ,
     ( T1 JOIN T2 ) AS T3 :
T3 WHERE XDURING OVERLAPS YDURING
```

This query gives a result with predicate "Supplier S# was able to supply part XP# throughout interval XDURING and part YP# throughout interval YDURING, and intervals XDURING and YDURING overlap." However, it fails to satisfy Condition *C1,* because the relation denoted by T3 and the final result both have two distinct stated-time attributes (in both cases, XDURING shows when supplier S# was able to supply part XP# and YDURING shows when supplier S# was able to supply part YP#). *Note:* In fact, this query also fails to satisfy Condition *C3*. And if we were to add a final step, in which (say) interval YDURING is effectively replaced by an interval expressed in terms of hours instead of days, then it would fail to satisfy Condition *C4* also.

Now let us turn to Condition *C2:*

C2: The result has at least one additional column, over and above any valid- or transaction-time column.

Condition *C2* clearly derives from the same facts as does Condition *C3*. And here is a **Tutorial D** query that fails to satisfy the condition, *mutatis mutandis:*

```
WITH ( SP_DURING { S#, DURING } ) AS T1 ,
      ( USING DURING * S_DURING MINUS T1 * ) AS T2 :
( T2 WHERE S# = S#('S1') ) { DURING }
```

This query gives intervals during which supplier S1 was unable to supply any parts at all. *Note:* Relvar S_DURING shows which suppliers were under contract when.

So much for the four conditions that characterize the "very special case" that, it is claimed, TSQL2 deals with very simply by means of its special "tables with temporal support" (together with certain other features, not yet discussed). Of course, we have not yet shown how queries are formulated in TSQL2 at all (though we will, in Section 6). Nevertheless, some obvious questions suggest themselves right away:

- How often do we need to formulate queries that do not fit the profile described above? Quite frequently, we believe.
- Even if most queries do fit that profile, is the claimed simplification worth all of the accompanying complexity?——in particular, is it worth jettisoning the relational model for? We do not believe it is.

- And in any case, do the simplifications actually work? We believe not (see Section 8).

5. TEMPORAL UPWARD COMPATIBILITY

In the previous section, we discussed what we called "the central idea" behind the TSQL2 language. However, the design of the language was also strongly motivated by another important idea (related to that previous one) called **temporal upward compatibility**. That idea can be described in outline as follows:

- Suppose we have some nontemporal database *D,* together with a set of applications that run against the database.
- Suppose we now want *D* to evolve to include some temporal support.
- Then it would be nice if we could just "add" that temporal support in such a way that those existing applications can continue to run correctly and unchanged against that temporal version of *D*.

If we meet this goal, then we say we have achieved *temporal upward compatibility* (hereinafter abbreviated, occasionally, to just TUC).

By way of example, suppose the nontemporal database shown in Fig. 3 is somehow converted into a fully temporal counterpart, such that all of the information in that database at the time of conversion is retained but is now timestamped in some manner that would allow all of the information shown in Fig. 4 to be recorded. *Note:* We very deliberately show the fully temporal counterpart in Fig. 4 in proper relational form, in order to simplify certain subsequent explanations that we need to make. In TSQL2, of course, the DURING attributes would be replaced by unnamed hidden columns, the resulting tables would be named just S and SP, not S_DURING and SP_DURING, and they would not in fact be proper relations at all.

S

S#
S1
S2
S3
S4
S5

SP

S#	P#
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

Fig. 3: A nontemporal database

S_DURING

S#	DURING
S1	[*d04*:*d10*]
S2	[*d02*:*d04*]
S2	[*d07*:*d10*]
S3	[*d03*:*d10*]
S4	[*d04*:*d10*]
S5	[*d02*:*d10*]

SP_DURING

S#	P#	DURING
S1	P1	[*d04*:*d10*]
S1	P2	[*d05*:*d10*]
S1	P3	[*d09*:*d10*]
S1	P4	[*d05*:*d10*]
S1	P5	[*d04*:*d10*]
S1	P6	[*d06*:*d10*]
S2	P1	[*d02*:*d04*]
S2	P1	[*d08*:*d10*]
S2	P2	[*d03*:*d03*]
S2	P2	[*d09*:*d10*]
S3	P2	[*d08*:*d10*]
S4	P2	[*d06*:*d09*]
S4	P4	[*d04*:*d08*]
S4	P5	[*d05*:*d10*]

Fig. 4: A temporal counterpart of Fig. 3

Then the conversion to temporal form, however it is carried out, is said to *achieve temporal upward compatibility* if and only if every operation that applied to the database before the conversion:

a. Still applies after the conversion, and

b. Has the same effect as before (apart, possibly, from effects that might become noticeable only by subsequent use of new operators that become available as a result of the conversion).

In order to illustrate this notion, suppose the temporal conversion has indeed been carried out, somehow; suppose further that the converted form of relvar SP is then updated in such a way that it now represents, somehow, exactly the information depicted in Fig. 4; and consider the effect of evaluating the following simple **Tutorial D** expression:

```
SP
```

Clearly, there are just two possibilities: Either the result is exactly as shown as the value of relvar SP in Fig. 3——not relvar SP_DURING in Fig. 4!——or temporal upward compatibility has not been achieved.

By way of a second example, suppose we perform the following DELETE on the temporal version of relvar SP (which we again assume represents the information shown as the value of relvar SP_DURING in Fig. 4):

```
DELETE SP WHERE S# = S#('S3') AND P# = P#('P2') ;
```

After this DELETE, if TUC is to be achieved, then the result of evaluating the expression

```
SP { S# }
```

on day 10 must not include supplier S3, because (as Fig. 3 shows) part P2 was the only part supplier S3 was currently——i.e., on day 10——able to supply before the DELETE. By contrast, suppose we have some way, after the temporal conversion, of expressing the query "Who was able to supply some part on day 9?" Then the result of that query on day 10 *must* include supplier S3. (In other words, the effect of the DELETE might be regarded, loosely, as replacing the value [*d08*:*d10*] of "attribute" DURING in the "tuple" for supplier S3 and part P2 by the value [*d08*:*d09*]. Remember, however, that in TSQL2 we cannot really explain the effect of the DELETE in this way, because in TSQL2 "relvar" SP does not really include a DURING "attribute," and "tuples" in that "relvar" thus do not really include a DURING component.)

More on terminology: The previous paragraph made use of the term "currently." Unfortunately, we now have to say that we do not find the meaning of that term very clear (in a TSQL2 context, that is), for reasons we now explain:

- Observe first of all that there seems to be a tacit assumption pervading TSQL2 to the effect that a temporal database will contain "historical relvars only" (to use the terminology of reference [6])——there is no suggestion that horizontal decomposition should be performed (yielding separate current and historical relvars), as in our own preferred approach.

(Our examples in this paper are in line with this assumption; in particular, note the appearance of the *d99* "end-of-time markers" in the hidden valid-time column in Fig. 2 in Section 3.) Thus, whatever *current* information the TSQL2 database contains will in fact be bundled in with those "historical relvars."

- Following on from the previous point: There is no suggestion that vertical decomposition should be performed, either. As a consequence, TSQL2 tables will typically not be in sixth normal form, 6NF [6]. Indeed, the recommended TSQL2 approach of simply "adding temporal support" to an existing nontemporal table——see Section 7 or the annotation to reference [15]——virtually guarantees that most TSQL2 tables will not be in 6NF. Of course, it is a large part of the point of TUC that it should be possible just to "add temporal support" to an existing table, but then the consequences of having to deal with non6NF tables must be faced up to. The TSQL2 literature has little or nothing to say on this issue.

- Now, the TSQL2 literature frequently refers to the conceptual representation of a temporal database as a *time series*——in other words, as a chronologically ordered sequence of entries, in which each entry is the value or "state" of the database at a particular point in time. However, the last or most recent entry in that time series is then typically referred to as *the current state*——a fact that, it might be argued, tends to suggest that beliefs about the future, such as the belief that a certain supplier's contract will terminate on some specific future date, cannot be expressed using the "temporal support" of "tables with temporal support" (?).

- More to the point: While the time-series notion might be conceptually agreeable (since it is clear that one possible representation of that time series is one involving intervals), surely the TSQL2 specification should state exactly which of those intervals are considered to contain the current time. But it does not.

- Indeed, the actual time referred to by the phrase "the current time" varies over time (of course!). So, if *S* is the set of all intervals that are considered to contain the current time, does *S* itself vary over time? If so, then many serious questions arise (some of which are discussed in reference [6]).

In connection with the foregoing, it is possibly relevant to note that reference [13] proposed the following definition for inclusion in the SQL standard: "The current valid-time state of a table with valid-time support is the valid-time state of that table at valid-time CURRENT_TIMESTAMP" (of course, the value of CURRENT_TIMESTAMP——a niladic built-in function in the SQL standard [5]——certainly does vary with time). By contrast, certain

examples in reference [15] seem to assume that any interval *i* such that END(*i*) = "the end of time" is one that contains the current time, regardless of the value of BEGIN(*i*).

Back to temporal upward compatibility. The TUC goal is TSQL2's justification for its special kinds of tables, with their hidden columns. For that goal would clearly not be achieved if (e.g.) converting the original nontemporal relvar SP to a temporal counterpart required the addition of an explicit new column——e.g., via an SQL ALTER TABLE statement, as here:

```
ALTER TABLE SP ADD COLUMN DURING ... ;
```

(Throughout this paper we follow "direct SQL" [5,8] in using semicolons as SQL statement terminators.)

Why would the TUC goal not be achieved? Because, of course, after execution of the foregoing ALTER TABLE statement, the result of the SQL expression

```
SELECT * FROM SP
```

would include column DURING as well as the S# and P# columns it would have included before the temporal conversion, thereby violating TUC. It follows that the conversion process cannot simply involve the addition of explicit new columns. See Section 7 for further discussion.

One last point: We have deliberately been somewhat vague as to the nature of the operations for which the TUC concept applies (or is even possible). The fact is, it is not at all clear whether it applies to——for example——all possible data definition operations, or dynamic SQL operations, etc. Here is what reference [15] has to say on the matter:

> "**Temporal upward compatibility:** An [SQL:1992] ... query, modification, view, assertion, [or] constraint ... will have the same effect on an associated snapshot database as on the temporal counterpart of the database."

(The expression "snapshot database" as used here simply means a regular nontemporal database.)

6. CURRENT, SEQUENCED, AND NONSEQUENCED OPERATIONS

Suppose now that the process of converting the database to temporal form, however it has to be done in order to achieve temporal upward compatibility, has in fact been done. Then TSQL2 supports three kinds of operations against such a database, which it calls *current, sequenced,* and *nonsequenced* operations, respectively. Briefly, if we regard the database as a time series once again, then we can characterize the three kinds of operations (loosely) as follows:

- **Current** operations apply just to the most recent entry in that time series. (The term *current* derives from the fact that such operations are intended to apply to current data.)

- **Sequenced** operations apply to all of the entries in that time series.[7] (The term *sequenced* derives from the fact that such operations are intended to apply to the entire "temporal sequence," or in other words "at every point in time.")

- **Nonsequenced** operations apply to some specified subset of the entries in that time series. (It is not clear why such operations are said to be *nonsequenced*. It might or might not help to point out that an operation that is not sequenced is not necessarily nonsequenced; likewise, one that is not nonsequenced is not necessarily sequenced. What is more, some operations are both sequenced and nonsequenced——though it is not possible to have an operation that is both current and sequenced or both current and nonsequenced.)

By way of example, consider the valid-time table shown in Fig. 5. Recall our assumption that today is day 10. Then——*very* loosely speaking——current operations are performed in terms of just those rows of that table whose hidden valid-time component includes day 10; sequenced operations are performed in terms of all of the rows; and nonsequenced operations are performed in terms of just those rows whose hidden valid-time component includes some day or days specified explicitly when the operator in question is invoked.

S

S#	
S1	[*d01*:*d01*]
S1	[*d05*:*d06*]
S2	[*d02*:*d04*]
S2	[*d06*:*d99*]
S3	[*d05*:*d99*]
S4	[*d03*:*d99*]
S6	[*d02*:*d03*]
S6	[*d06*:*d09*]

Fig. 5: A TSQL2 valid-time table

Current Operations

[7] A slight oversimplification; actually, it is possible to restrict sequenced operations (like nonsequenced operations) to apply to some specified subset of the entries in that time sequence.

Current operations are, of course, precisely those operations that were available before the conversion to temporal form; temporal upward compatibility requires those operations still to be available and to have the same effect as they did before the conversion. A *current query* involves the execution of some current operation of a retrieval nature; a *current modification* involves the execution of some current operation of an updating nature. Of course, current modifications must now have some additional effects "behind the scenes" (as it were), over and above those effects that are directly visible to the user of the current modification in question. For example, consider again this DELETE example from Section 5:

```
DELETE FROM SP WHERE S# = S#('S3') AND P# = P#('P2') ;
```

(We have added the keyword FROM in order to make the DELETE into a valid SQL statement. Also, we assume, here and throughout this paper, that expressions of the form S#('S*x*') and P#('P*y*') are valid SQL literals, of types S# and P#, respectively.)

If table SP is a valid-time table, with current value the TSQL2 analog of the SP_DURING value shown in Fig. 4, then the logical effect of the foregoing DELETE must be to do both of the following:

a. To delete the row for supplier S3 and part P2, as requested (since that row's valid-time component does include day 10, "the current date")

b. To insert a row into "the history portion" of the table for supplier S3 and part P2 with a valid time of [*d08*:*d09*]

In practice, of course, the deletion and subsequent insertion could probably be combined into a single row replacement.

Sequenced Operations

We turn now to sequenced and nonsequenced operations——sequenced ones in this subsection and nonsequenced ones in the next. After the conversion to temporal form has been performed and updates have been applied to the temporal version, the database will include historical as well as current data. Thus, the question arises as to how that historical data can be accessed. Clearly, the answer to this question is going to involve some new operations that were not available before the conversion, and those new operations are, precisely, the sequenced and nonsequenced operations already mentioned. *Note:* As usual, we take the term *access* to include both query and modification operations. For reasons of brevity and simplicity, however, we will have little to say in this paper regarding modifications, either sequenced or nonsequenced.

TSQL2 uses "statement modifiers" to specify both sequenced and nonsequenced operations (the term is a misnomer, actually, since it is not always statements *per se* that such modifiers modify).

Those modifiers take the form of prefixes that can be attached to certain statements and certain (table-valued) expressions. We can summarize the available prefixes, and the rules regarding the operand table(s) and the result table, if any, as follows:

Prefix	Operand(s)	Result
none	*any*	*nhc*
VALIDTIME	VT *or* BT	VT
TRANSACTIONTIME	TT *or* BT	TT
VALIDTIME AND TRANSACTIONTIME	BT	BT
NONSEQUENCED VALIDTIME	VT *or* BT	*nhc*
NONSEQUENCED TRANSACTIONTIME	TT *or* BT	*nhc*
VALIDTIME AND NONSEQUENCED TRANSACTIONTIME	BT	VT
NONSEQUENCED VALIDTIME AND TRANSACTIONTIME	BT	TT
NONSEQUENCED VALIDTIME AND NONSEQUENCED TRANSACTIONTIME	BT	*nhc*

Explanation: The abbreviations VT, TT, and BT stand for a valid-time table, a transaction-time table, and a bitemporal table, respectively; the abbreviation *nhc* stands for "no hidden columns" (in other words, the table in question is just a regular SQL table). For example, we can see that if the prefix NONSEQUENCED VALIDTIME is used, then every operand table must be either a valid-time table or a bitemporal table, and the result (if the statement or expression to which the prefix applies in fact returns a result) is a regular table. Note that the result has a hidden valid-time column only if a prefix specifying VALIDTIME (without NONSEQUENCED) is specified, and a hidden transaction-time column only if a prefix specifying TRANSACTIONTIME (without NONSEQUENCED) is specified.

At this point, a couple of minor oddities arise:

- First, the prefix (e.g.) NONSEQUENCED VALIDTIME is regarded in the TSQL2 literature as specifying a *valid-time nonsequenced* operation, not a nonsequenced valid-time operation. Although we find this inversion of the modifiers a trifle illogical, we will conform to it in what follows.

- Second, observe that nonsequenced operations involve the explicit keyword NONSEQUENCED, but sequenced operations do not involve any explicit SEQUENCED counterpart; for example, a *sequenced valid-time* operation is specified by just the prefix VALIDTIME, unadorned.

For simplicity, let us concentrate on sequenced valid-time operations specifically, until further notice. Let *X* be an expression or statement that is syntactically valid on the nontemporal version of the database. Let every table mentioned in *X* map to a counterpart with valid-time support in the temporal version of the database. Then VALIDTIME *X* is an expression or statement that

a. Is syntactically valid on the temporal version of the database, and

b. Is conceptually evaluated against that temporal database *at every point in time*.

Each such conceptual evaluation is performed on a nontemporal database that is derived from the temporal one by considering just those rows whose associated valid times include the applicable point in time. The results of those conceptual evaluations are then conceptually combined by a process analogous to packing to yield the overall result. *Note:* Perhaps we should say rather that the combination process is *somewhat* analogous to packing; as we will see a little later, that overall result is in fact not precisely defined. But let us ignore this point for now.

By way of illustration, consider first the *current* DELETE example from the subsection above entitled "Current Operations":

```
DELETE FROM SP WHERE S# = S#('S3') AND P# = P#('P2') ;
```

As you will recall, the effect of this DELETE (ignoring the side-effect of "inserting into the historical record") is to delete just the fact that supplier S3 is *currently* able to supply part P2. However, if we prefix the DELETE statement with the modifier VALIDTIME, as here——

```
VALIDTIME
DELETE FROM SP WHERE S# = S#('S3') AND P# = P#('P2') ;
```

——then the effect is now to delete *all* rows showing supplier S3 as able to supply part P2 from the valid-time table SP, no matter what the associated valid times might be. (In terms of the data values in Fig. 4, the effect is to delete the fact that supplier S3 was able to supply part P2 throughout the interval from day 8 to day 10——but it might delete more than that, if there were any other rows for supplier S3 and part P2.)

Analogously, the TSQL2 expression

```
VALIDTIME
SELECT * FROM SP
```

returns the "real" value of the valid-time table SP, hidden valid-time column and all. Note carefully, however, that that hidden column remains hidden in the result; in fact, a valid-time sequenced query always returns a valid-time table (i.e., a table with a hidden valid-time column and *no* hidden transaction-time column). See the final subsection in this section for a discussion of how such hidden columns can be accessed in that result (or in any other table with temporal support, of course).

Incidentally, observe that the expression SELECT * FROM SP is indeed an expression and not a statement. The foregoing example thus illustrates our earlier claim that "statement modifier" is really a misnomer.

Here now are a couple more examples of valid-time sequenced queries:

```
VALIDTIME                          |    VALIDTIME
SELECT DISTINCT S# FROM SP         |    SELECT DISTINCT S# FROM S
                                   |    EXCEPT
                                   |    SELECT DISTINCT S# FROM SP
```

(In the first example, we assume table S has valid-time support; in the second, we assume tables S and SP both have valid-time support.) These expressions are TSQL2 formulations for two sample queries——or, rather, TSQL2 counterparts to those queries——that we used as a basis for introducing the temporal database problem in reference [6]:

- Get S#-DURING pairs for suppliers who have been able to supply at least one part during at least one interval of time, where DURING designates a maximal interval during which supplier S# was in fact able to supply at least one part.
- Get S#-DURING pairs for suppliers who have been unable to supply any parts at all during at least one interval of time, where DURING designates a maximal interval during which supplier S# was in fact unable to supply any part at all.

The first expression results in a table showing supplier numbers for suppliers who have ever been able to supply anything, paired in the hidden valid-time column with the maximal intervals during which they have been able to do so. The second expression is analogous. Note carefully, however, that those "maximal intervals" are indeed still represented by hidden columns; if we want to access those hidden columns——as surely we will?——we will have to make use of the operators described in the final subsection of this section (see below). Note too that we are being slightly charitable to TSQL2 here! In fact, the proposals of reference [13] did not explicitly require the result of a query like the ones illustrated above to satisfy any such "maximality" condition. What is more, they did not impose any other

requirement in place of such a condition, either; as a consequence, the actual value of an expression such as VALIDTIME SELECT DISTINCT S# FROM SP is not precisely specified (it is not even clear whether the inclusion of the keyword DISTINCT has any effect). See Section 11 for further discussion.

Suppose now that S and SP are tables with *transaction*-time support. Then the prefix TRANSACTIONTIME can be used in place of VALIDTIME in examples like those shown above; the operations in question then become *transaction-time sequenced* operations (transaction-time sequenced *queries* specifically, in all of those examples except the very first). A transaction-time sequenced query returns a transaction-time table (i.e., a table with a hidden transaction-time column and *no* hidden valid-time column).

Finally, suppose S and SP are bitemporal tables. Then the prefix VALIDTIME AND TRANSACTIONTIME can be used, in which case the operations in question become (prosaically enough) *valid-time sequenced and transaction-time sequenced* operations. A valid-time sequenced and transaction-time sequenced query returns a bitemporal table. *Note:* If the result of such a query is indeed automatically packed, it is pertinent to ask whether they are packed on valid time first and then transaction time or the other way around. The literature does not appear to answer this question.

Nonsequenced Operations

Nonsequenced operations are specified by means of the prefixes NONSEQUENCED VALIDTIME and NONSEQUENCED TRANSACTIONTIME. Furthermore, if the operand tables are bitemporal, then all possible combinations——e.g., (sequenced) VALIDTIME AND NONSEQUENCED TRANSACTIONTIME——are available. Thus, operations on bitemporal tables can be simultaneously sequenced with respect to valid time and nonsequenced with respect to transaction time, or the other way around, or sequenced with respect to both, or nonsequenced with respect to both.

Here is an example of a nonsequenced query:

```
NONSEQUENCED VALIDTIME
SELECT DISTINCT P# FROM SP
```

Table SP must have valid-time support in order for this query to be legal. The result is a table with no hidden valid-time column at all, representing part numbers for all parts we currently believe ever to have been available from any supplier.

Despite the somewhat arcane prefixes, nonsequenced operations are comparatively easy to understand, for here TSQL2 is effectively reverting to regular SQL semantics. Well, almost——there is a glitch![8] The glitch is that "regular

[8] At least, there is according to reference [15], but not (or possibly not) according to reference [13]. See Example 14 in Section 8.

semantics" implies that we should be able to reference the hidden columns in the regular way; but such references are impossible, precisely because the columns are hidden. We therefore need some special mechanism in order to access the hidden columns. In TSQL2, that mechanism is provided by the operators VALIDTIME(*T*) and TRANSACTIONTIME(*T*)——see the subsection immediately following. *Note:* Orthogonality dictates that these operators be available in connection with current and sequenced operations too, despite the fact that we have introduced them in the context of a discussion of nonsequenced operations specifically. However, the effect of including such operators in such queries is unclear to the present writers.

Accessing the Hidden Columns

Consider the following example (note in particular the VALIDTIME operator invocations):

```
NONSEQUENCED VALIDTIME
SELECT T1.S# AS X#, T2.S# AS Y#,
       BEGIN ( VALIDTIME ( T2 ) ) AS SWITCH_DATE
FROM   S AS T1, S AS T2
WHERE  END ( VALIDTIME ( T1 ) ) = BEGIN ( VALIDTIME ( T2 ) )
```

This expression returns a table *without* any hidden valid-time column in which each row gives a pair of supplier numbers X# and Y# and a date such that, on that date, supplier X#'s contract terminated and supplier Y#'s contract began (according to our current belief). The expression is the TSQL2 analog of the following **Tutorial D** query (expressed in terms of the database of Fig. 4):

```
WITH ( ( ( S RENAME ( S# AS X#, DURING AS XD ) )
           JOIN
         ( S RENAME ( S# AS Y#, DURING AS YD ) ) )
       WHERE END ( XD ) = BEGIN ( YD ) ) AS T1 ,
     ( EXTEND T1 ADD ( BEGIN ( YD ) AS SWITCH_DATE ) ) AS T2 :
T2 { X#, Y#, SWITCH_DATE }
```

Of course, the operator invocation VALIDTIME(*T*) is valid in TSQL2 only if the table denoted by *T* has valid-time support; likewise, the operator invocation TRANSACTIONTIME(*T*) is valid only if the table denoted by *T* has transaction-time support. Observe, incidentally, how these operators implicitly rely on the fact that any given TSQL2 table has at most one hidden valid-time column and at most one hidden transaction-time column.

7. DATA DEFINITION STATEMENTS

We now consider the effects of the ideas discussed in the foregoing sections on the SQL CREATE TABLE and ALTER TABLE statements.

Valid-Time Tables

There are two ways to create a valid-time base table in TSQL2. The underlying principle in both cases is just to extend a nontemporal counterpart of the table in question by "adding valid-time support," both to that counterpart as such and to the constraints——primary and foreign key constraints in particular——that apply to that counterpart. "Adding valid-time support" can be done either directly in the original CREATE TABLE statement or subsequently by means of appropriate ALTER TABLE statements.

By way of example, consider the following CREATE TABLE statements, which will suffice to create a TSQL2 counterpart of the database of Fig. 3 (note the text in boldface):

```
CREATE TABLE S ( S# S#,
       VALIDTIME PRIMARY KEY ( S# ) )
       AS VALIDTIME PERIOD ( DATE ) ;

CREATE TABLE SP ( S# S#, P# P#,
       VALIDTIME PRIMARY KEY ( S#, P# ),
        VALIDTIME FOREIGN KEY ( S# ) REFERENCES S )
       AS VALIDTIME PERIOD ( DATE ) ;
```

Explanation:

- The specification AS VALIDTIME ... (in line 3 of the CREATE TABLE for suppliers and line 4 of the CREATE TABLE for shipments) indicates that tables S and SP are tables with valid-time support; i.e., they have hidden valid-time columns. They are not packed on those columns (perhaps because such packing could lead to a violation of temporal upward compatibility, if the AS VALIDTIME ... specification appeared in an ALTER TABLE——rather than CREATE TABLE, as here——and the table in question currently contained any duplicate rows).
- The specification PERIOD (DATE) following AS VALIDTIME gives the data type for the hidden valid-time columns; PERIOD is a "type constructor" (it is the TSQL2 counterpart of our INTERVAL type generator), and DATE is the corresponding point type.[9] *Note:* TSQL2 could not use the keyword INTERVAL here, because the SQL standard already uses that keyword for something else. More to the point, observe that——of course——*any* TSQL2 table, regardless of whether or not it has any kind of "temporal support," can have any number of regular

[9] Note that TSQL2 follows its keyword PERIOD with the name of the point type in parentheses, whereas we follow our keyword INTERVAL with the name of the point type attached by means of an underscore instead. A related observation is that the TSQL2 analog of what we would express as, e.g., INTERVAL_DATE ([*di*:*dj*]) is just PERIOD ([*di*:*dj*]); in other words, TSQL2 assumes the type of the interval——or period, rather——can be inferred from the type of the begin and end points *di* and *dj*. We do not agree with this latter position, for reasons explained in detail in reference [6].

columns of some period type. As noted near the end of Section 3, TSQL2 regards such columns as containing what it calls *user-defined time* [10].

- The VALIDTIME prefixes on the primary key and foreign key specifications specify that the corresponding constraints are *valid-time sequenced constraints*. Moreover:
  - A VALIDTIME PRIMARY KEY constraint is analogous, somewhat, to a WHEN / THEN constraint as defined in reference [6] (except that we do not believe in the idea of being forced to single out some specific candidate key and make it "primary," and as a matter of fact neither does SQL). It is not clear whether TSQL2 allows a VALIDTIME PRIMARY KEY constraint to coexist with a regular PRIMARY KEY constraint, though it is clear that the existence of a VALIDTIME one makes a regular one more or less redundant.
  - A VALIDTIME FOREIGN KEY constraint is analogous, again somewhat, to a "foreign U_key" constraint as defined in reference [6]. Note that the referenced table——S, in our example——must have valid-time support in order for the VALIDTIME FOREIGN KEY constraint to be valid.

  Absence of the VALIDTIME prefix on a primary or foreign key specification, in the presence of AS VALIDTIME, means the corresponding constraint is a *current* one; that is, it applies only to those rows whose valid-time component is considered to contain the current time (speaking rather loosely).

Suppose now, in contrast to the foregoing, that the *non*temporal tables S and SP have already been defined, thus:

```
CREATE TABLE S ( S# S#,
       PRIMARY KEY ( S# ) ) ;

CREATE TABLE SP ( S# S#, P# P#,
       PRIMARY KEY ( S#, P# ),
       FOREIGN KEY ( S# ) REFERENCES S ) ;
```

Suppose further that we now wish to "add valid-time support" to these tables (remember the goal of temporal upward compatibility). Then the following more or less self-explanatory ALTER TABLE statements will suffice (again, note the text in boldface):

```
ALTER TABLE S ADD VALIDTIME PERIOD ( DATE ) ;
ALTER TABLE S ADD VALIDTIME PRIMARY KEY ( S# ) ;

ALTER TABLE SP ADD VALIDTIME PERIOD ( DATE ) ;
ALTER TABLE SP ADD VALIDTIME PRIMARY KEY ( S#, P# ) ;

ALTER TABLE SP ADD VALIDTIME FOREIGN KEY ( S#) REFERENCES S ;
```

In rows that already exist when the valid-time support is added, the new (hidden) column is set to contain a period of the

form [*b*:*e*], where *b* is the time of execution of the ALTER TABLE and *e* is "the end of time."[10] Whether it is necessary to drop the primary and foreign keys that were defined for the tables before the valid-time support was added is unclear.

Transaction-Time Tables

Creation of transaction-time base tables is similar but not completely analogous to the creation of valid-time base tables:

```
CREATE TABLE S ( S# S#,
       TRANSACTIONTIME PRIMARY KEY ( S# ) )
       AS TRANSACTIONTIME ;

CREATE TABLE SP ( S# S#, P# P#,
       TRANSACTIONTIME PRIMARY KEY ( S#, P# ),
       TRANSACTIONTIME FOREIGN KEY ( S# ) REFERENCES S )
       AS TRANSACTIONTIME ;
```

The AS TRANSACTIONTIME specifications are more or less self-explanatory; observe, however, that no data type is specified, because (as mentioned in a footnote in Section 4) transaction times in TSQL2 are always of a data type that is chosen by the DBMS. The TRANSACTIONTIME prefixes on the primary and foreign key specifications are analogous to their VALIDTIME counterparts——except that there seems to be no point in having them (although they *are* permitted), because the corresponding *current* constraints must surely imply that these *transaction-time sequenced constraints* are always satisfied. (By definition, transaction times cannot be updated; it therefore follows that constraints that apply to the current state of affairs must apply equally to the historical record, since everything in that historical record must once have been current.) Also, if the prefix is omitted on a foreign key specification, then the referenced table can be of any kind (not necessarily even one with temporal support); in every case, the constraint is then treated as a current constraint rather than a transaction-time sequenced one.

Adding transaction-time support to existing tables via ALTER TABLE is analogous to its valid-time counterpart. In particular, in rows that already exist when the transaction-time support is added, the new (hidden) column is apparently set to the same initial value as it is in the case of adding valid time——i.e., it is set to a period of the form [*b*:*e*], where *b* is the time of execution of the ALTER TABLE and *e* is "the end of time"——even though neither the *b* value nor the *e* value seems to make any sense (the *b* value is clearly incorrect, and the *e* value means we have

[10] Actually, reference [13] says *e* is *the immediate predecessor* of "the end of time," but this is surely just a slip, probably arising from confusion over notation (in effect, confusing [*b*:*e*] with [*b*:*e*)——see reference [6]). Reference [15] says it is "the end of time."

transaction times that refer to the future). We omit further discussion here.

Bitemporal Tables

Finally, here are the CREATE TABLE statements needed to create bitemporal versions of tables S and SP:

```
CREATE TABLE S ( S# S#,
       VALIDTIME AND TRANSACTIONTIME PRIMARY KEY ( S# ) )
       AS VALIDTIME PERIOD ( DATE ) AND TRANSACTIONTIME ;

CREATE TABLE SP ( S# S#, P# P#,
       VALIDTIME AND TRANSACTIONTIME PRIMARY KEY ( S#, P# ),
       VALIDTIME AND TRANSACTIONTIME FOREIGN KEY ( S# )
                                     REFERENCES S )
       AS VALIDTIME PERIOD ( DATE ) AND TRANSACTIONTIME ;
```

These statements should once again be self-explanatory.

Adding bitemporal support to existing tables via ALTER TABLE is analogous to its valid-time and transaction-time counterparts. We omit further discussion here.

8. STATEMENT MODIFIERS ARE FLAWED

This brings us to the end of our brief overview of TSQL2 basics. In this section and the next three, we give our reasons for rejecting the TSQL2 approach, and indeed for seriously questioning its very motivation. *Note:* Those reasons are very similar to those that have previously been aired in the international standards and academic research communities, precisely because two of the authors of reference [6] (Darwen and Lorentzos) have been at the forefront in articulating such objections in those communities.

The goal of the present section is to demonstrate a number of logical problems with the basic idea of statement modifiers. In order to meet that goal, we present a series of simple examples that illustrate those problems. The examples are numbered for purposes of subsequent reference. Here then is the first example (a current query against versions of tables S and SP with——let us assume——valid-time support):[11]

```
1. SELECT DISTINCT S.S#, SP.P#
   FROM   S, SP
   WHERE  S.S# = SP.S#
   AND    SP.P# = P#('P1')
```

Note: It might be objected that this first example is not a very sensible one, inasmuch as (a) the result of the query will

[11] All of the examples in this section are based on a certain simple combination of a join, a restriction, and a projection. Consideration of examples involving something a little more complicated is left as an exercise.

have part number P1 in every row and (b) the DISTINCT cannot possibly have any effect. However, the example is adequate as a basis for illustrating the points we wish to make, and we will stay with it.

It is easy to see that the following reformulation (Example 2) is guaranteed under all circumstances to yield the same result as Example 1:

2.
```
SELECT DISTINCT S.S#, T1.P#
FROM   S, ( SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1
WHERE  S.S# = T1.S#
```

Now consider the TSQL2 valid-time counterpart of Example 1:

3.
```
VALIDTIME
SELECT DISTINCT S.S#, SP.P#
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = P#('P1')
```

The obvious question arises as to whether simply adding the VALIDTIME prefix to Example 2 gives an equivalent reformulation, as it did before:

4.
```
VALIDTIME
SELECT DISTINCT S.S#, T1.P#
FROM   S, ( SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1
WHERE  S.S# = T1.S#
```

The answer to this question is *no!*——in fact, the putative reformulation is syntactically invalid. The reason is that, in the presence of the VALIDTIME modifier, each and every "table reference" in the FROM clause is required to denote a table with valid-time support, and in the example the second such reference in the outer FROM clause does not do so; as you can see, in fact, that second reference involves an expression of the form SELECT * FROM SP WHERE SP.P# = P#('P1'), and that expression lacks the statement modifier that is needed to make it yield a table with valid-time support. In order to obtain the correct desired reformulation, therefore, we must insert the VALIDTIME prefix in more than one place, as here:

5.
```
VALIDTIME
SELECT DISTINCT S.S#, T1.P#
FROM   S, ( VALIDTIME
            SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1
WHERE  S.S# = T1.S#
```

Note, by the way, that the foregoing quirk arises (at least in part) because of an existing quirk in SQL: The first table reference in the outer FROM clause (i.e., S), does not require the prefix, simply because a simple table name like S does not constitute a valid query in SQL! If we were to replace it by, for example, the expression SELECT * FROM S (which is a valid query,

of course), then we would have to include the prefix as well, as here:

```
 6. VALIDTIME
    SELECT DISTINCT T2.S#, T1.P#
    FROM ( VALIDTIME
           SELECT * FROM S ) AS T2,
         ( VALIDTIME
           SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1
    WHERE  T2.S# = T1.S#
```

What if the table denoted by a table reference in a FROM clause happens to be a view? Suppose, for example, that view VS is defined as follows:

```
 7. CREATE VIEW VS AS
           SELECT * FROM S ;
```

In principle——and in SQL too, normally——a reference to a given view and the expression that defines that view are logically interchangeable. The question therefore arises as to whether we can replace the expression SELECT * FROM S in the outer FROM clause in Example 6 by a reference to VS, as follows:

```
 8. VALIDTIME
    SELECT DISTINCT T2.S#, T1.P#
    FROM ( VALIDTIME VS ) AS T2,
         ( VALIDTIME
           SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1
    WHERE  T2.S# = T1.S#
```

Again the answer is *no,* and again the replacement gives rise to a syntax error, because VS is not a table with valid-time support (and simply inserting the VALIDTIME prefix does not make it one, either). Instead, we have to place that prefix *inside the view definition:*

```
 9. CREATE VIEW VS AS
           VALIDTIME
           SELECT * FROM S ;
```

VS is now a table with valid-time support and a reference to it can appear wherever a reference to S can appear.

A similar observation applies when VS is defined "inline," using a WITH clause:

```
10. WITH VS AS ( SELECT * FROM S )
    VALIDTIME
    SELECT DISTINCT T2.S#, T1.P#
    FROM ( VALIDTIME
           SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1,
           VS AS T2
    WHERE  T2.S# = T1.S#
```

This expression is again invalid. However, it can be rescued by placing the VALIDTIME prefix inside the WITH clause:


```
11. WITH VS AS ( VALIDTIME SELECT * FROM S )
    VALIDTIME
    SELECT DISTINCT T2.S#, T1.P#
    FROM ( VALIDTIME
           SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1,
           VS AS T2
    WHERE  T2.S# = T1.S#
```

In fact, according to reference [13], in a query that includes a WITH clause, the VALIDTIME prefix cannot be placed at the beginning of the entire expression. Rather, it can only be placed as shown above, between the WITH clause and the main body of the expression.

It follows from all of the foregoing that the TSQL2 claim to the effect that a temporal counterpart of a nontemporal query can be easily obtained by just adding a prefix is not entirely valid and needs to be made more precise. For example, consider the following nontemporal query:

```
12. WITH VS AS ( SELECT * FROM S )
    SELECT DISTINCT T2.S#, T1.P#
    FROM ( SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1,
           VS AS T2
    WHERE  T2.S# = T1.S#
```

We cannot obtain a temporal counterpart of this query by just adding a VALIDTIME prefix to the beginning, nor, as we have already seen, can we do so by just adding it in the middle, between the WITH clause and the main body. Rather, we have to add it three times, as shown in Example 11.

Now, all of the examples we have shown so far have made use just of the VALIDTIME prefix and have dealt just with tables with valid-time support. As you would probably expect, however, the whole discussion is applicable in like manner to the TRANSACTIONTIME prefix and tables with transaction-time support. Here, for instance, is a bitemporal counterpart of Example 11 (and here we must assume that S and SP are bitemporal tables):

```
13. WITH VS AS ( VALIDTIME AND TRANSACTIONTIME
                 SELECT * FROM S )
    VALIDTIME AND TRANSACTIONTIME
    SELECT DISTINCT T2.S#, T1.P#
    FROM ( VALIDTIME AND TRANSACTIONTIME
           SELECT * FROM SP WHERE SP.P# = P#('P1') ) AS T1,
           VS AS T2
    WHERE  T2.S# = T1.S#
```

However, the whole discussion appears not to be applicable in like manner in connection with prefixes that use the NONSEQUENCED modifier! For example, suppose we take Example 5 and replace both of the VALIDTIME prefixes by the prefix NONSEQUENCED VALIDTIME:

14. NONSEQUENCED VALIDTIME
    SELECT DISTINCT S.S#, T1.P#
    FROM   S, (NONSEQUENCED VALIDTIME
                SELECT * FROM SP WHERE SP.P# = P#('P1')) AS T1
    WHERE S.S# = T1.S#

This expression is syntactically invalid, because the second table reference in the outer FROM clause denotes a table without temporal support. (In fact, it is not clear exactly what table it does denote; references [13] and [15] contradict each other on the issue. Details of just how they contradict each other are beyond the scope of this paper, however.)

The net of all of the foregoing is as follows. First, the suppliers table S is (according to our original assumption) a table with valid-time support, from which it follows that in TSQL2 the expression S can appear as a table reference in a FROM clause in an SQL query that has the VALIDTIME prefix. However, the expression SELECT * FROM S yields a result that is not a table with valid-time support, and so that expression cannot appear as a table reference in a FROM clause in such a query. By contrast, the expression VALIDTIME SELECT * FROM S can so appear. But the expression VS, when defined to mean the same as the expression SELECT * FROM S, cannot so appear, and nor can the expression VALIDTIME VS——nor, for that matter, can the expression VALIDTIME SELECT * FROM VS. Taken together, these anomalies show that TSQL2 fails to meet normal expectations of a computer language with respect to construction of expressions from subexpressions and replacement of subexpressions by introduced names.

But there is still more to be said regarding introduced names. All such names we have considered so far have resulted from view definitions and WITH clauses. If we go on to consider introduced names that result from user-defined functions, we encounter even more serious problems, problems that make us conclude that the concept of statement modifiers as manifested in TSQL2 is *fundamentally flawed*. By way of example, consider first the following query (again we assume that tables S and SP have valid-time support):

15. VALIDTIME
    SELECT S.S#
    FROM   S
    WHERE S.S# NOT IN (SELECT SP.S# FROM SP)

The overall result of this query will obviously depend on whether the VALIDTIME prefix applies to the whole expression, including the parenthesized subexpression following the IN operator, or whether it applies only to the portion of the query *not* included in those parentheses:

a. *(VALIDTIME applies to whole expression)* The result is a valid-time table in which the hidden valid-time column indicates, for each supplier number, a time interval

throughout which the supplier in question was unable to supply any parts.

b. *(VALIDTIME applies only to outer portion)* The result is a valid-time table in which the hidden valid-time column indicates, for each supplier number, a time interval throughout which the supplier in question was not among those suppliers who are *currently* able to supply any parts.

It is clear from many examples in the TSQL2 literature that the first of these two interpretations is the one intended. Yet it is difficult to obtain a reading of the expression that is consistent with that interpretation, because the table denoted by the parenthesized subexpression seems, according to our understanding of unprefixed expressions in TSQL2, to give just supplier numbers of suppliers *currently* able to supply some part.[12] Clearly, we must revise that understanding somehow, perhaps by replacing that "currently" by something like "at the relevant point in time." (We are deferring here to the notion that a TSQL2 sequenced query is conceptually evaluated at each point in time, with subsequent packing——or, rather, some unspecified variant of packing——of the resulting conceptual sequence of results.)

Although the foregoing revised understanding is very vague, it can presumably be made more precise, somehow, and so we have probably not dealt a mortal blow, yet, to the idea of statement prefixing. But let us see where else this example might lead us. We now consider the possibility of replacing the expression in the WHERE clause——S.S# NOT IN (SELECT SP.S# FROM SP)——by an equivalent invocation of a user-defined function. The function in question could be defined in SQL as follows:

```
16. CREATE FUNCTION UNABLE_TO_SUPPLY_ANYTHING ( S# S# )
            RETURNS BOOLEAN
            RETURN ( S# NOT IN ( SELECT SP.S# FROM SP ) ) ;
```

Given this function, the following expression——

```
17. SELECT S.S#
    FROM   S
    WHERE  UNABLE_TO_SUPPLY_ANYTHING ( S.S# )
```

——is clearly equivalent to this one:

```
18. SELECT S.S#
    FROM   S
    WHERE  S.S# NOT IN ( SELECT SP.S# FROM SP )
```

The natural question to ask now is whether the following expression——

[12] What is more, it is our further understanding that that table has no hidden valid-time column; as a consequence, it is not clear how the comparisons implied by the IN operator can be the ones that TSQL2 seems to want, either.


```
19. VALIDTIME
    SELECT S.S#
    FROM   S
    WHERE  UNABLE_TO_SUPPLY_ANYTHING ( S.S# )
```

——is equivalent to this one:

```
20. VALIDTIME
    SELECT S.S#
    FROM   S
    WHERE  S.S# NOT IN ( SELECT SP.S# FROM SP )
```

The answer to this question is far from clear! Indeed, the following excerpt from the concluding summary of reference [1] is perhaps revealing in this connection:

> Implementing functions ... is another interesting research topic. Specifically, function calls are affected by statement modifiers, so that the semantics of a function call will depend on whether it is used in a temporal upward-compatible, a sequenced, or a nonsequenced context.

The authors of reference [1] appear to be claiming here that Examples 19 and 20 are equivalent. In that case, we have to ask why, if "function calls are affected by statement modifiers," the same is not true of references to view names and names introduced using WITH. But in any case the authors are also clearly admitting that "function calls" in the context of statement modifiers have not been fully researched. We venture to think that anybody attempting to undertake that research is likely to meet with insuperable problems. If the body of the function consists of some highly complex series of statements, including assignments, branches, and exception-handling operations, how is the function to be conceptually evaluated at each point in time other than by *actually* evaluating it at each point in time? *Note:* The matter is made even worse in SQL specifically by the fact that user-defined functions can be coded in programming languages other than SQL, using the same facilities (such as embedded SQL, SQL/CLI, or JDBC) as are available to client application programs.

One last point to close this section: It might be thought that the "U_" operators of reference [6] suffer from the same problems as TSQL2's statement modifiers, since those operators also involve a prefix that affects the semantics of the expression following that prefix. However, we believe——of course!——that the same criticisms do not apply. The reason is that our prefixes are defined very carefully to affect *only the outermost operator* in the pertinent expression (and just what it is that constitutes that "pertinent expression" is well-defined, too, both syntactically and semantically). If that operator is monadic, then it is precisely the single relation operand to that monadic operator that is initially unpacked; if the operator is dyadic, then it is precisely the two relation operands to that dyadic operator that are initially unpacked. In both cases, the regular

relational operation is then performed on the unpacked operand(s), and the result is then packed again. In brief: Our USING prefixes can be thought of as *operator* modifiers, not *statement* (or, rather, expression) modifiers.

9. CONSEQUENCES OF HIDDEN COLUMNS

If, as we believe, the concept of statement modifiers is irredeemably flawed, then perhaps nothing more needs to be said. As we have seen, however, TSQL2 also involves a radical departure from *The Information Principle*. Just to remind you, that principle states that all information in the database should be represented in one and only one way: namely, by means of relations. (SQL tables are not true relations, of course, but for the sake of the present discussion we can pretend they are; that is, we can overlook for present purposes such matters as duplicate rows, nulls, and left-to-right column ordering. What we want to do is consider the differences between *TSQL2* tables——rather than SQL tables in general——and true relations.)

Now, the uniformity of *structure* provided by adherence to *The Information Principle* carries with it uniformity of *mode of access* and uniformity of *description:* All data in a table is accessed by reference to its columns, using column names for that purpose; also, to study the structure of a table, we have only to examine the description (as recorded in the catalog) of each of its columns.

TSQL2's departure from this uniformity leads to several complications of the kind that the relational model was explicitly designed to avoid. For example, new syntax is needed (as we have seen) for expressing temporal queries and modifications; new syntax is also needed for referencing hidden columns; new features are needed in the catalog in order to describe tables with temporal support; and similar new features are needed in the "SQL descriptor areas" used by generalized applications that support *ad hoc* queries [5,8]. These are not trivial matters, as the discussions of earlier sections in this paper should have been sufficient to demonstrate.

It is worth taking a moment to elaborate on the implications of hidden columns for generalized applications (the final complication in the list called out in the previous paragraph). Consider the tasks that are typically performed by such an application. A simple example is the task of saving the result of an arbitrary query *Q*. So long as *Q* is well-formed, in the sense that every result column has a unique name, then all the application has to do is create an SQL table *T,* taking its definition from the column names and data types given in the SQL descriptor area for the query, and then execute the SQL statement INSERT INTO *T Q*. Now consider, by contrast, what the application will have to do if the query *Q* happens to take one of the forms

illustrated by the examples in the previous section. The simple solution that worked so well before will clearly now be very far from adequate.

10. LACK OF GENERALITY

TSQL2's support for tables with temporal support and temporal intervals fails to include support for operations on intervals in general. Of course, it does support some of the operators normally defined for intervals in general——BEGIN, MEETS, OVERLAPS, UNION, and so on (though we have not discussed these operators in this paper)——but even in the case of temporal intervals it fails to provide any counterpart of the useful shorthands we have described in reference [6] for operations on relations and relvars involving interval attributes. In particular, it has nothing equivalent to the PACK and UNPACK operators,[13] nor to any of the "U_" operators, nor to any of the proposed shorthands for constraints ("U_key" constraints and others) or for updating.

TSQL2 lacks generality in another sense, too. If it is reasonable to use hidden columns for valid times and transaction times, would it not be equally reasonable to use hidden columns for other kinds of data? For example, consider a requirement to record measurements showing variation in soil acidity at various depths [9]. If we can have tables with valid-time support, should we not also be able to have, analogously, tables with valid-depth support, tables with valid-pH support, and perhaps tables with both valid-depth and valid-pH support? In fact, is there any reason to confine such facilities to hidden *interval* columns? Perhaps relvar SP in the nontemporal version of suppliers and shipments could be a table with valid-P# support (having S# as its only regular column), or a table with valid-S# support (having P# as its only regular column). Such observations might raise a smile, but we offer them for serious consideration. The fact is, as soon as we permit the first deviation from *The Information Principle,* we have opened the door——possibly the floodgates——to who knows what further indiscretions to follow.

Incidentally, lest we be accused of possible exaggeration in the foregoing, we would like to draw your attention to another extract from reference [1]. The authors are discussing "interesting directions for future research":

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

[13] ATSQL [1] does have an analog of our PACK operator.

11. IMPRECISE SPECIFICATION

Consider again Example 3 from Section 8:

```
3. VALIDTIME
   SELECT DISTINCT S.S#, SP.P#
   FROM   S, SP
   WHERE  S.S# = SP.S#
   AND    SP.P# = P#('P1')
```

We have already mentioned (in Section 6) TSQL2's failure to specify precisely what set of rows constitutes the result of a valid-time or transaction-time query. The response usually given to this complaint is that if all of the tables in some set of tables are equivalent, in the sense that they yield the same result when unpacked (or packed) on some interval column, and one table in that set is agreed to be a correct result for a given query, then any table in that set is equally correct and can be produced as the result of the query. In other words, if tables S and SP represent precisely the information shown for relvars S and SP in Fig. 4 in Section 5, then either of the tables shown in Fig. 6——as well as literally billions of others[14]——might be produced as the result of the query shown above as Example 3. (It might help to point out explicitly that the table on the left-hand side of the figure is packed on the hidden valid-time column.)

S#	P#	
S1	P1	[*d04*:*d10*]
S2	P1	[*d02*:*d04*]
S2	P1	[*d08*:*d10*]

S#	P#	
S1	P1	[*d04*:*d08*]
S1	P1	[*d06*:*d07*]
S1	P1	[*d05*:*d10*]
S1	P1	[*d04*:*d09*]
S2	P1	[*d02*:*d03*]
S2	P1	[*d04*:*d04*]
S2	P1	[*d08*:*d10*]

Fig. 6: Two possible results for Example 3

But the foregoing position is surely unacceptable. The various results that are regarded as equally correct under the given equivalence relationship are distinguishable from one another in SQL. Even the cardinality, unless it happens to be zero (or possibly one), is not constant over all of those results!

[14] Actually the upper bound is infinite, since SQL tables can have duplicate rows. Even if we ignore duplicate rows, however, the number of possible results is still astronomical; in the case at hand, for example, there are over 137,438,953,472 such possible results——and this figure is just a lower bound. (In case you are interested, an upper bound for the same example is over a trillion——1,099,511,627,776, to be precise.)

It follows that, in general, TSQL2's temporal queries (and modifications too, presumably) are *indeterminate*.

That said, the problem can easily be addressed by specifying, for example, some suitably packed form to be the actual result. Therefore, we do not regard this fault in TSQL2, astonishing though it is, as in itself militating against the whole approach. We think the other reasons we have given are sufficient to do that.

12. CONCLUDING REMARKS

We have presented a brief overview and analysis of TSQL2 and found much in it to criticize. In fact, we have two broad (and orthogonal) sets of criticisms: one having to do with the overall approach in general, and one having to do with the language's poor fit with SQL specifically (even more specifically, with certain of the features that were added in SQL:1999——for example, triggers, row types, and "typed tables"). In this paper, we have concentrated on the first of these two sets of criticisms; for a discussion of the second, see references [2], [3], and [4].

By way of conclusion, we repeat in summary form some of our biggest criticisms from the first category.

- *Regarding "the central idea":* Even if we accept for the sake of argument that TSQL2 succeeds in its objective of simplifying the formulation of queries that satisfy Conditions *C1-C4,* it is surely obvious that there are many, many queries that fail to satisfy those conditions.
- *Regarding temporal upward compatibility:* Here we reject the very idea that the goal might be desirable, let alone achievable. In particular, we reject the idea that just "adding temporal support" is a good way to design temporal databases, because (a) it leads to the bundling of current and historical data, and (b) it leads to relvars (tables) that are not in 6NF. Further, we reject the notion that "current operations" should work exactly as they did before, because that notion leads to the notion of hidden columns and (apparently) to the notion of statement modifiers.
- *Regarding statement modifiers:* We have discussed at great length (in Section 8) our reasons for believing this concept to be logically flawed. Furthermore, we do not believe it can be fixed.
- *Regarding hidden columns:* We have discussed this issue at considerable length, too. Hidden columns are a logical consequence of the objective of temporal upward compatibility——but they constitute the clearest possible violation of *The Information Principle,* and they lead directly to many of TSQL2's other problems.

REFERENCES AND BIBLIOGRAPHY

1. Michael H. Böhlen, Christian S. Jensen, and Richard T. Snodgrass: "Temporal Statement Modifiers," *ACM TODS 25,* No. 4 (December 2000).

2. Hugh Darwen: "Valid Time and Transaction Time Proposals: Language Design Aspects," in reference [7].

3. Hugh Darwen, Mike Sykes, *et al*.: "Concerns about the TSQL2 Approach to Temporal Databases," Kansas City, Mo. (May 1996); *ftp:// sqlstandards.org/SC32/WG3/Meetings/MCI_1996_05_ KansasCity_USA/mci071.ps*.

   A precursor to reference [4]. As explained in Section 5, a major part of the rationale for TSQL2 was *temporal upward compatibility* (TUC). Briefly, TUC means that it should be possible to convert an existing nontemporal database into a temporal one by just "adding temporal support," and then have existing nontemporal applications still run (and run correctly) against the now temporal database. Among other things, the present paper (i.e., reference [3]) raises questions as to whether TUC is even a sensible goal, and some of the arguments it makes in this connection are worth summarizing here. Consider the following example. Suppose we start with an SQL table EMP, with columns EMP#, DEPT#, and SALARY. Suppose we now "add valid time support" to that table as described in Section 7, so that every existing EMP row is timestamped with the valid-time value (a *period* or interval value) "from now till the end of time." But:

   - The table is not yet telling the truth (as reference [3] puts it), since, in general, existing employees did not join the company or move to their current department or reach their current salary "now." So those valid-time timestamps all need to be updated, somehow.
   - The table is also not yet telling the truth in that it contains rows only for current employees and current department assignments and current salary levels. All of the historical information for previous employees and previous departments and previous salaries needs to be added, somehow.
   - We cannot tell for any given row whether the timestamp shows when the employee moved to the indicated department, or when the employee reached the indicated salary, or perhaps even when the employee joined the company. It is thus likely that the table will need to be vertically decomposed into three separate tables (one each for employment history, department history, and salary history), as explained in reference [6].

- What would happen if——as is not at all unlikely——table EMP already included columns DATE_OF_JOINING_DEPT and DATE_OF_LAST_INCREASE before the "valid-time support" was added?
- Even if all of the foregoing issues can be addressed successfully, we are left with tables that represent both history and the current state of affairs. It is thus likely that each such table will need to be split into two, as described in reference [6].

The net effect of the foregoing points (it seems to us) is that

a. Converting a nontemporal database to a temporal counterpart involves——necessarily——much more than just "adding temporal support,"

and hence that, in general,

b. Having existing applications run unchanged after such a conversion is not a very realistic goal.

4. Hugh Darwen, Mike Sykes, *et al*.: "On Proposals for Valid-Time and Transaction-Time Support," Madrid, Spain (January 1997); *ftp://sqlstandards.org/SC32/WG3/Meetings/MAD_1997_01_Madrid_ESP/mad146.ps*.

   The "UK response" to reference [13].

5. C. J. Date and Hugh Darwen: *A Guide to the SQL Standard* (4th edition). Reading, Mass.: Addison-Wesley (1997).

6. C. J. Date, Hugh Darwen, and Nikos A. Lorentzos: *Temporal Data and the Relational Model*. San Francisco, Calif.: Morgan Kaufmann (2003).

   The present paper was originally prepared as an appendix to this book, though it has been edited to make it stand by itself as far as possible.

7. Opher Etzion, Sushil Jajodia, and Suryanaryan Sripada (eds.): *Temporal Databases: Research and Practice*. New York, N.Y.: Springer-Verlag (1998).

   This book is an anthology giving "the state of the temporal database art" as of about 1997. It is divided into four major parts, as follows:

   1. Temporal Database Infrastructure
   2. Temporal Query Languages
   3. Advanced Applications of Temporal Databases
   4. General Reference

8. International Organization for Standardization (ISO): *Database Language SQL,* Document ISO/IEC 9075:1999. Also available as American National Standards Institute (ANSI) Document ANSI NCITS.135-1999.

9. Nikos A. Lorentzos and Vassiliki J. Kollias: "The Handling of Depth and Time Intervals in Soil Information Systems," *Comp. Geosci. 15,* 3 (1989).

10. Richard Snodgrass and Ilsoo Ahn: "A Taxonomy of Time in Databases," Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Texas (May 1985).

    The source of the terms *transaction time, valid time,* and *user-defined time*. *Note:* Transaction time and valid time are discussed at length in reference [6], but "user-defined time" is not. Reference [10] defines this term to mean temporal values and attributes that are "not interpreted by the DBMS"; examples are date of birth, date of last salary increase, or time of arrival. Observe, however, that in the approach to temporal databases espoused and described in reference [6], transaction times and valid times are also——like all other values and attributes!——"not interpreted by the DBMS." While it might make sense to have a term for "times" that are neither transaction times nor valid times, the idea that "user-defined times" are *operationally* different from the others makes sense only if we start by assuming a nonrelational approach to the temporal database problem in the first place.

11. R. T. Snodgrass *et al*.: "TSQL2 Language Specification," *ACM SIGMOD Record 23,* No. 1 (March 1994).

12. Richard T. Snodgrass (ed.): *The TSQL2 Temporal Query Language*. Norwell, Mass.: Kluwer Academic Publishers (1995).

13. Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner: "Adding Valid Time to SQL/Temporal" and "Adding Transaction Time to SQL/Temporal," Madrid, Spain (January 1997); *ftp://sqlstandards.org/SC32/WG3/Meetings/MAD_1997_01_Madrid_ESP/mad146.ps*.

14. Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner: "Transitioning Temporal Support in TSQL2 to SQL3," in reference [7].

15. Richard T. Snodgrass: *Developing Time-Oriented Database Applications in SQL*. San Francisco, Calif.: Morgan Kaufmann (2000).

    The following remarks on temporal database design are taken from this reference (we find them interesting inasmuch as they describe an approach that is diametrically opposite to that advocated by the present authors in reference [6]): "In the approach that we espouse here, conceptual design initially ignores the time-varying nature of the applications. We focus on capturing the *current reality* and temporarily ignore any history that may be useful to capture. This selective amnesia somewhat simplifies what is often a highly complex task of

capturing the full semantics ... An added benefit is that existing conceptual design methodologies apply in full ... Only after the full design is complete do we augment the ER [= *entity/relationship*] schema with ... time-varying semantics ... Similarly, logical design proceeds in two stages. First, the nontemporal ER schema is mapped to a nontemporal relational schema, a collection of tables ... In the second stage, each of the [temporal annotations on the otherwise nontemporal ER schema] is applied to the logical schema, modifying the tables or integrity constraints to accommodate that temporal aspect."

***** End *** End *** End *****

