Managing Temporal Data

A Five-Part Series

Richard T. Snodgrass

September 3, 1998

TR-28

A TIMECENTER Technical Report

Title	**Managing Temporal Data**
	A Five-Part Series
Author(s)	Richard T. Snodgrass
Publication History	June – October 1998. Appeared in *Database Programming and Design*. September, 1998. A TIMECENTER Technical Report.

TIMECENTER Participants

Contents

Abstract

Temporal data is pervasive, and challenging to manage in SQL. The June through October issues of *Database Programming and Design* (volume 11, issues 6–10) included a special series on temporal databases; the five articles in that series are reproduced here. Three separate case studies: a neonatal intensive care unit, a commercial cattle feed yard, and astronomical star catalogs, were used to illustrate how temporal applications can be implemented in SQL. The concepts of valid time versus transaction time and of current, sequenced and nonsequenced integrity constraints, queries, and modifications were emphasized.

1 Of Duplicates and Septuplets

This special series explores the many issues that arise when attempting to define and manage time-varying data. Such data is pervasive. It has been estimated that one of every 50 lines of database application code involves a date or time value. Data warehouses are by definition time-varying: Ralph Kimball states that every data warehouse has a time dimension. Often the time-oriented nature of the data is what lends it value.

DBAs and application programmers constantly wrestle with the vagaries of such data. They find that overlaying simple concepts, such as duplicate prevention, on time-varying data can be surprisingly subtle and complex. And they are perplexed that trade publications and books do not provide guidance and techniques for handling such data.

The five articles in this series will address this need by presenting specific, easily applied ways to manage time-varying data, generally in SQL. Each will include concrete examples of code that can be immediately used in ongoing development efforts. Equally important, we will introduce and illustrate new ways to think about temporal data, imposing structure on a messy topic.

In honor of the McCaughey children, the world's only known set of living septuplets, this first article will consider duplicates, of which septuplets are just a novel special case.

Specifically, we examine the ostensibly simple task of preventing duplicate rows, via a constraint in a table definition. Preventing duplicates using SQL is thought to be trivial, and truly is, when the data is not time-varying. But when history is retained, things get much dicier. In fact, over such data several interesting kinds of duplicates can be defined. And, as is so often the case, the most relevant kind is the hardest to prevent, and requires an aggregate or a complex trigger! We'll first use standard SQL-92, then delve into the machinations required when using DB2, Oracle and Sybase.

1.1 Twin Time

As I write this on January 6, 1998, Kenneth Robert McCaughey, the first to be born and the biggest, is now at home; he was released three days ago. The other six are still listed in fair condition, but are expected to come home by the end of the month. (Full information is available on, of course, the septuplets' web page: www.mccaugheyseptuplet.com) We consider here a NICUStatus table recording the status of patients in the Neonatal Intensive Care Unit at Blank Children's Hospital in Des Moines, Iowa, an excerpt of which is shown in Figure 1.

In this *temporal* table, we made several design decisions, which will be elaborated on in future columns. Each row, indicating the condition of an infant, is timestamped with a pair of SQL DATEs. The from_date column indicates the day the child first was listed at that status. The to_date column indicates the day the child's condition changed. In concert, these columns specify a *period* over which the status was valid. Tables can be timestamped with values other than periods. This representation of the period is termed *closed-open*, because the starting date is contained in the period but the ending date is not. Periods can also be represented in other ways, though it turns out that the closed-open representation is the best choice. We denote a row that is currently valid with a to_date of *forever*, which in SQL-92 is 9999-12-31 (thereby introducing a year-9999 problem...) Alternative approaches to *now* are certainly possible. Some use the NULL value to denote "now"; others use a date way in the past, such as 1988. Finally, we mention in passing that this table represents the status *in reality*, termed *valid time*; there exist other useful kinds of time. For *valid-time* tables, the timestamp is termed the *period of validity*. (Here, the notion of "timestamp" is entirely separate from the SQL-92 column type TIMESTAMP.)

Such tables are very common in practice. Often there are many columns, with the timestamp of a row indicating when that combination of values was valid.

A duplicate in the SQL sense is a row that exactly matches, column for column, another row. We will term such duplicates *nonsequenced* duplicates, for reasons that will become clear shortly. The last two rows of the above table

Name	Status	from_date	to_date
Kenneth Robert	serious	1997-11-19	1997-11-21
Alexis May	serious	1997-11-19	1997-11-27
Natalie Sue	serious	1997-11-19	1997-11-25
Kelsey Ann	serious	1997-11-19	1997-11-26
Brandon James	serious	1997-11-19	1997-11-26
Nathan Roy	serious	1997-11-19	1997-11-28
Joel Steven	critical	1997-11-19	1997-11-20
Joel Steven	serious	1997-11-20	1997-11-26
Kenneth Robert	fair	1997-11-21	1998-01-03
Alexis May	fair	1997-11-27	1998-01-11
Alexis May	fair	1997-12-02	9999-12-31
Alexis May	fair	1997-12-02	9999-12-31

Figure 1: Excerpt from the NICUStatus table

Name	Status
Alexis May	fair
Alexis May	fair
Alexis May	fair

Figure 2: Current snapshot of the NICUStatus table

are nonsequenced duplicates. However, there are three other kinds of duplicates that are interesting, all present in this table. These variants arise due to the temporal nature of the data.

The last three rows are *value-equivalent*, in that the values of all the columns except for those of the timestamp are identical. Value equivalence is a particularly weak form of duplication. It does, however, correspond to the traditional notion of duplicate for a non-time-varying, *snapshot*, table, e.g., with only the two columns, Name and Status.

The last three rows are also current duplicates. A *current* duplicate is one present in the *current timeslice* of the table. As *now* is January 6, 1998, the current timeslice of the above table is simply as shown in Figure 2. Interestingly, whether a table contains current duplicate rows can change over time, even if no modifications are made to the table. In a week, one of of these current duplicates will quietly disappear.

The most useful variant is a *sequenced* duplicate. The adjective *sequenced* means that the constraint is applied independently at every point in time. The last three rows are sequenced duplicates. These rows each state that Alexis was in fair condition for most of December 1997 and the first eleven days of 1998.

The following table indicates how these variants interact. Each entry specifies whether rows satisfying the variant in the left column will also satisfy the variant listed across the top. A check mark states that the top variant will be satisfied; an empty entry states that it may not. For example, if two rows are non-sequenced duplicates, they will also be sequenced duplicates, for the entire period of validity. However, two rows that are sequenced duplicates are not necessarily nonsequenced duplicates, as illustrated by the second-to-last and last rows of the example temporal table.

	Sequenced	Current	Value-equivalent	Nonsequenced
Sequenced	$\surd$		$\surd$	
Current	$\surd$	$\surd$	$\surd$	
Value-equivalent			$\surd$	
Nonsequenced	$\surd$		$\surd$	$\surd$

The least restrictive form of duplication is value equivalence, as it simply ignores the timestamps. Note from above that this form implies no other. The most restrictive is nonsequenced duplication, as it requires all the column values to match exactly. It implies all but current duplication.

SQL's UNIQUE constraint prevents value-equivalent rows.

```
CREATE TABLE NICUStatus (
    Name CHAR(15),
    Status CHAR(8),
    from_date DATE,
    to_date DATE,
    UNIQUE (Name, Status)
)
```

Intuitively, a value-equivalent duplicate constraint states that "once a condition is assigned to a patient, it can never be repeated later," because doing so would result in a value-equivalent row.

We can also use a UNIQUE constraint to prevent non-sequenced duplicates, by simply including the timestamp columns.

```
CREATE TABLE NICUStatus (
    ...
  UNIQUE (Name, Status, from_date, to_date)
)
```

While nonsequenced duplicates are easy to prevent via SQL statements, such constraints are not that useful in practice. The intuitive meaning of the above nonsequenced unique constraint is something like, "a patient cannot have a condition twice over identical periods." However, this constraint can be satisfied by simply shifting one of the rows a day earlier or later, so that the periods of validity are not identical; it is still the case that the patient has the same condition at various times.

Preventing current duplicates involves just a little more effort.

```
CREATE TABLE NICUStatus (
    ...
  CHECK (NOT EXISTS (SELECT N1.SSN
                     FROM NICUStatus AS N1
                     WHERE 1 < (SELECT COUNT(Name)
                        FROM NICUStatus AS N2
                        WHERE N1.Name = N2.Name AND N1.Status = N2.Status
                           AND N1.from_date <= CURRENT_DATE
                           AND CURRENT_DATE < N1.to_date
                           AND N2.from_date <= CURRENT_DATE
                           AND CURRENT_DATE < N2.to_date)))
)
```

Here the intuition is that no patient can (currently) have two identical status values, or equivalently, "each patient has at most one status." The present tense is used to indicate "at the current time."

As mentioned above, the problem with a current uniqueness constraint is that it can be satisfied today, but violated tomorrow, even if there are no changes made to the underlying table.

If we *know* that the application will never store future data, we can approximate a current uniqueness constraint by simply including the to_date column in the UNIQUE constraint.

```
CREATE TABLE NICUStatus (
    ...
  UNIQUE (Name, Status, to_date)
)
```

This works because all current data will have the same to_date value, the special value DATE '9999-12-31'.

Preventing sequenced duplicates is similar to preventing current duplicates. Operationally, two rows are sequenced duplicates if they are value equivalent and their periods of validity overlap. This definition is equivalent to the one given above.

```
CREATE TABLE NICUStatus (
    ...
  CHECK (NOT EXISTS (SELECT N1.Name
                FROM NICUStatus AS N1
                WHERE 1 < (SELECT COUNT(Name)
                  FROM NICUStatus AS N2
                  WHERE N1.Name = N2.Name AND N1.Status = N2.Status
                    AND N1.from_date < N2.to_date AND N2.from_date < N1.to_date)))
)
```

The tricky last line just states that the periods of validity overlap.

The intuition behind a sequenced uniqueness constraint is that "at no time can a patient have two identical conditions." This constraint is a natural one. A sequenced constraint is the logical extension of a conventional constraint on a nontemporal table.

1.2 Vendor Implementations

None of the above SQL-92 code fragments work in DB2, for a variety of reasons. First, DB2 (version 2.1.2) doesn't support UNIQUE. Instead, a primary key constraint must be used. However, this constraint requires that all of the columns so indicated be designated as not null.Preventing value-equivalent rows can be expressed in DB2 as follows.

```
CREATE TABLE NICUStatus (
    Name CHAR(15) NOT NULL,
    Status CHAR(8) NOT NULL,
    from_date DATE,
    to_date DATE,
    PRIMARY KEY (Name, Status)
)
```

The same trick can be used to present nonsequenced duplicates or current duplicates.

Secondly, check constraints in DB2 cannot contain subqueries. Unfortunately, several of the above statements require those pesky subqueries. We can express such constraints as triggers. The check constraint to prevent sequenced duplicates can be implemented as an insert trigger (an analogous update trigger would also be required).

```
CREATE TRIGGER seq_duplicates
AFTER INSERT ON NICUStatus FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT N1.Name
            FROM NICUStatus AS N1
            WHERE 1 < (SELECT COUNT(Name)
                    FROM NICUStatus AS N2
                    WHERE N1.Name = N2.Name AND N1.Status = N2.Status
                      AND N1.from_date < N2.to_date AND N2.from_date < N1.to_date)))
SIGNAL SQLSTATE '75000' ('Name and Status must be sequenced unique' )
```

Oracle supports the UNIQUE constraint. However, Oracle (version 7.3.2) doesn't allow tables to be mentioned in check constraints, so preventing current or sequenced duplicates require triggers. Things are a little easier in Oracle, as a single trigger can be defined for both insert and update, and the aggregate can be avoided in Oracle via the rowid facility.

```
CREATE OR REPLACE TRIGGER seq_duplicates
AFTER INSERT OR UPDATE ON NICUStatus
DECLARE
  valid INTEGER;
BEGIN
  SELECT 1
  INTO valid
  FROM DUAL
  WHERE NOT EXISTS (SELECT N1.Name
        FROM NICUStatus N1, NICUStatus N2
        WHERE N1.Name = N2.Name AND N1.Status = N2.Status
          AND N1.from_date < N2.to_date AND N2.from_date < N1.to_date
          AND N1.rowid != N2.rowid);

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR ( -20001, 'Name and Status must be sequenced unique' );
END;
```

In this trigger, if the where clause is not satisfied (i.e., if there *is* a sequenced duplicate), then the exception will be raised, causing the transaction to abort. (DUAL is a dummy system table provided by Oracle for exactly this kind of situation.)

Sybase's syb_identity function can also be used to differentiate rows, if "auto identity" is enabled. This is analogous to Oracle's rowid facility.

The moral of the story is, adding the timestamp columns to the UNIQUE clause will prevent nonsequenced duplicates, value-equivalent duplicates, or some forms of current duplicates, which unfortunately is rarely what is desired. The natural temporal generalization of a conventional duplicate on a snapshot table is a sequenced duplicate. To prevent sequenced duplicates, a rather complex check constraint, or even one or more triggers, is required.

As a challenge, to presage the topic of a future article, consider specifying in SQL a primary key constraint on a period-stamped valid-time table. Then try specifying a referential integrity constraint between two period-stamped valid-time tables. It *is* possible, but is certainly not easy.

In the next installment, I'll examine ways to query valid-time tables. We'll see strong parallels to the material here, as I describe current, sequenced, and non-sequenced queries.

1.3 Temporal Databases

The accepted term for a database that records time-varying information is a "temporal database" [1]. The term "time-varying" database is awkward, because even if only the current state is kept in the database (e.g., the current stock, the current salary and job title of employees), this database will change as reality changes, and so could perhaps be considered a time-varying database. The term "historical database" implies that the database only stores "historical" information, that is, information about the past; a temporal database may store information about the future, e.g., schedules or plans.

The official definition of temporal database is "a database that supports some aspect of time, not counting user-defined time" [1]. So, what is user-defined time? This is defined as "an uninterpreted attribute domain of date and time. User-defined time is parallel to domains such as 'money' and integer...It may be used for attributes such as 'birth day' and 'hiring date'." The intuition here is that adding a birthdate column to an employee table does not render it temporal, especially since the birthdate of an employee is presumably fixed, and applies to that employee forever. The presence of a DATE column will not a priori render the database a temporal database; rather, the database must record the time-varying nature of the enterprise it is modeling.

It is perhaps surprising that temporal databases is a very active area within database research. There have been some 1600 (!) papers written about this topic, over a 20-year period. The number of papers has been rising exponentially over this period; several hundred now appear each year. Many are included in the most recent bibliography, which has pointers to some five prior bibliographies [7]. [2] provides a brief survey of the field. The most complete book on the topic is Tansel et al. [6]; a more recent text provides an updated summary [8].

FDYD_ID	LOT_ID_NUM	PEN_ID	HD_CNT	FROM_DATE	TO_DATE
1	137	1	17	1998-02-07	1998-02-18
1	219	1	43	1998-02-25	1998-03-01
1	219	1	20	1998-03-01	1998-03-14
1	219	2	23	1998-03-01	1998-03-14
1	219	2	43	1998-03-14	9999-12-31
1	374	1	14	1998-02-20	9999-12-31

Table 1: The LOT_LOC table

2 Querying Valid-Time State Tables

A few days ago Oprah Winfrey won a legal suit brought by Texas cattlemen who had argued that Oprah had made false statements about U.S. beef in a 1996 television show on mad cow disease.

This case brought to mind the outbreak in the U.S. in the summer of 1997, long after the mad cow disease had caused so much concern in the U.K. Sixteen cases of people falling ill to a lethal strain of the bacterium Escherichia coli, E. coli O157:H7, all in Colorado, were eventually traced back to a processing plant in Columbus, Nebraska. The plant's operator, Hudson Foods, eventually recalled 25 million pounds of frozen hamburger to attempt to stem this outbreak.

That particular plant presses about 400,000 pounds of hamburger daily. Ironically, this plant received high marks for its cleanliness and adherence to Federal food processing standards. What lead to the recall of about one-fifth of the plant's annual output was the lack of data that could link particular patties back to the slaughterhouses that supply carcasses to the Columbus plant. It is believed that the meat was contaminated in only one of these slaughterhouses, but without such tracking, all were suspect.

Put simply, the lack of an adequate temporal database cost Hudson Foods over $20 million.

Dr. Brad De Groot is a veterinarian at the University of Nebraska at Lincoln, about 60 miles southeast of Columbus. He is also interested in improving the health maintenance of cows on their way to your freezer. He hopes to establish the temporal relationships between putative risk factor exposure (e.g., a previously healthy cow sharing a pen with a sick animal) and subsequent health events (e.g., the cow later succumbs to a disease). These relationships can lead to an understanding of how disease is transfered to and among cattle, and ultimately, to better detection and prevention regimes. As input to this epidemiologic study, he is massaging data from commercial feed yard record keeping systems to extract the movement of some 55,000 head of cattle through the myriad pens of several large feed yards in Nebraska.

These cattle are grouped into "lots", with subsets of lots moved from pen to pen. One of Brad's tables, the LOT_LOC table, records how many cattle from each lot are residing in each pen of each feed yard. The full schema for this table has nine columns.

LOT_LOC(FDYD_ID, LOT_ID_NUM, PEN_ID, HD_CNT, FROM_DATE, FROM_MOVE_ORDER,
TO_DATE, TO_MOVE_ORDER, RECORD_DATE)

This table is a "valid-time state table", in that it records information valid at some time, and it records states, that is, facts that are true over a period of time. The FROM and TO columns delimit the "period of validity" of the information in the row. The "temporal granularity" of this table is somewhat finer than a day, in that the move orders are sequential, allowing multiple movements in a day to be ordered in time. The RECORD_DATE identifies when this information was recorded; we will explain this kind of information in Section 4, below. For the present purposes, we will omit the FROM_MOVE_ORDER, TO_MOVE_ORDER, and RECORD_DATE columns, and express our queries on the simplified schema. The first four columns are integer columns; the last two are of type DATE.

In the above instance, 17 head of cattle were in pen 1 for 11 days, moving inauspiciously off the feed yard on February 18. 14 head of cattle from lot 374 are still in pen 1 (we use "forever" to denote currently valid rows), and 23 head of cattle from lot 219 were moved from pen 1 to pen 2 on March 1, with the remaining 20 head of cattle in that lot moved to pen 2 on March 14, where they still reside.

The previous section discussed three basic kinds of uniqueness assertions: current, sequenced, and nonsequenced. A current uniqueness constraint (of patient and status, on a table recording the status of patients in a neonatal intensive care unit) was exemplified with “each patient has at most one status condition,” a sequenced constraint with “at no time can a patient have two identical conditions,” and a nonsequenced constraint with “a patient cannot have a condition twice over identical periods.” We saw that the sequenced constraint was the most natural analog of the nontemporal constraint, yet was the most challenging to express in SQL. For the LOT_LOC table, the appropriate uniqueness constraint would be that FDYD_ID, LOT_ID_NUM, PEN_ID are unique at every time, which is a sequenced constraint.

2.1 Temporal Projection and Selection

These notions carry over to queries. In fact, for each conventional (non-temporal) query, there exist current, sequenced, and non-sequenced variants over the corresponding valid-time state table. Consider the non-temporal query, “How many head of cattle from lot 219 in feed yard 1 are in each pen?” over the non-temporal table LOT_LOC_SNAPSHOT(FDYD_ID, LOT_ID_NUM, PEN_ID, HD_CNT). Such a query is easy to write in SQL.

```
SELECT PEN_ID, HD_CNT
FROM LOT_LOC
WHERE FDYD_ID = 1 AND LOT_ID_NUM = 219
```

The current analog over the LOT_LOC valid-time state table is “How many head of cattle from lot 219 in yard 1 are (currently) in each pen?” For such a query, we only are concerned with currently valid rows, and we need only to add a predicate to the “where” clause asking for such rows.

```
SELECT PEN_ID, HD_CNT
FROM LOT_LOC
WHERE FDYD_ID = 1 AND LOT_ID_NUM = 219
  AND TO_DATE = DATE '9999-12-31'
```

This query returns the following result, stating that all the cattle in the lot are currently in a single pen.

PEN_ID	HD_CNT
2	43

The sequenced variant is “Give the history of how many head of cattle from lot 219 in yard 1 were in each pen.” This is also easy to express in SQL. For selection and projection (which is what this query involves), converting to a sequenced query involves merely appending the timestamp columns to the target list of the select statement.

```
SELECT PEN_ID, HD_CNT, FROM_DATE, TO_DATE
FROM LOT_LOC
WHERE FDYD_ID = 1 AND LOT_ID_NUM = 219
```

The result provides the requested history. We see that lot 219 moved around a bit.

PEN_ID	HD_CNT	FROM_DATE	TO_DATE
1	43	1998-02-25	1998-03-01
1	20	1998-03-01	1998-03-14
2	23	1998-03-01	1998-03-14
2	43	1998-03-14	9999-12-31

The non-sequenced variant is “How many head of cattle from lot 219 in yard 1 were, at some time, in each pen?” Here we don’t care when the data was valid. Note that the query doesn’t ask for totals; it is interested in whenever a portion of the requested lot was in a pen. The query is simple to express in SQL, as the timestamp columns are simply ignored.

```
SELECT PEN_ID, HD_CNT
FROM LOT_LOC
WHERE FDYD_ID = 1 AND LOT_ID_NUM = 219
```

PEN_ID	HD_CNT
1	43
1	20
2	23
2	43

Non-sequenced queries are often awkward to express in English, but can sometimes be useful.

2.2 Temporal Joins

Temporal joins are considerably more involved. Consider the non-temporal query "Which lots are co-resident in a pen?" Such a query could be a first step in determining exposure to putative risks. Indeed, the entire epidemiologic investigation revolves around such queries.

Again, we start by expressing the query on a hypothetical snapshot table, LOT_LOC_SNAPSHOT, as follows. The query involves a self-join on the table, along with projection and selection. The first predicate ensures we don't get identical pairs; the second and third predicates test for co-residency.

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID
FROM LOT_LOC_SNAPSHOT AS L1, LOT_LOC_SNAPSHOT AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
  AND L1.FDYD_ID = L2.FDYD_ID
  AND L1.PEN_ID = L2.PEN_ID
```

The current version of this query on the temporal table is constructed by adding a currency predicate (a TO_DATE of forever) for each correlation name in the from clause.

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
  AND L1.FDYD_ID = L2.FDYD_ID
  AND L1.PEN_ID = L2.PEN_ID
  AND L1.TO_DATE = DATE '9999-12-31'
  AND L2.TO_DATE = DATE '9999-12-31'
```

This query will return the empty table on the above data, as none of the lots are currently co-resident (lots 219 and 374 are currently in the feed yard, but in different pens).

The non-sequenced variant is "Which lots were in the same pen, perhaps at different times?" As before, non-sequenced joins are easy to specify: just ignore the timestamp columns.

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
  AND L1.FDYD_ID = L2.FDYD_ID
  AND L1.PEN_ID = L2.PEN_ID
```

The result is the following: all three lots had once been in pen 1.

L1	L2	PEN_ID
137	219	1
137	219	1
137	374	1
219	374	1
219	374	1

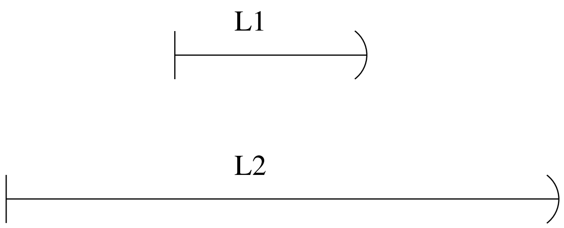
Note however that at no time were any cattle from lot 137 co-resident with either of the other two lots. To determine co-residency, the sequenced variant is used: "Give the history of lots being co-resident in a pen." This requires the cattle to actually be in the pen together, at the same time. The result of this query on the above table is the following.

L1	L2	PEN_ID	FROM_DATE	TO_DATE
219	374	1	1998-02-25	1998-03-01

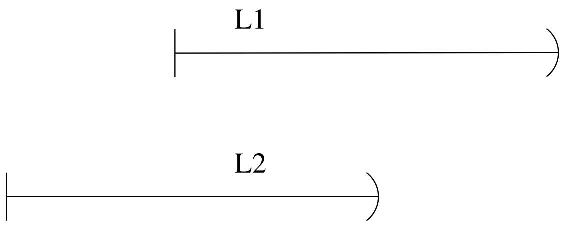
A sequenced join is somewhat challenging to express in SQL. We assume that the underlying table contains no (sequenced) duplicates, that is, a lot can be in a pen at most once at any time.

The sequenced join query must do a case analysis of how the period of validity of each row L1 of LOT_LOC overlaps the period of validity of each row L2, also of LOT_LOC; there are four possible cases.

In the first case, the period associated with the L1 row is entirely contained in the period associated with the L2 row. Since we are interested in those times when both lots are in the same pen, we compute the intersection of the two periods, which in this case is the contained period, that is, the period from L1.FROM_DATE to L1.TO_DATE. Below, we illustrate this case, with the right end emphasizing the closed-open representation.



In the second case, neither period contains the other, and the desired period is the intersection of the two periods of validity.



The other cases similarly identify the overlap of the two periods. Each case is translated to a separate select statement, because the target list is different in each case.

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID, L1.FROM_DATE, L1.TO_DATE
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
   AND L1.FDYD_ID = L2.FDYD_ID
   AND L1.PEN_ID = L2.PEN_ID
   AND L2.FROM_DATE <= L1.FROM_DATE
   AND L1.TO_DATE <= L2.TO_DATE
UNION
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID, L1.FROM_DATE, L2.TO_DATE
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
   AND L1.FDYD_ID = L2.FDYD_ID
   AND L1.PEN_ID = L2.PEN_ID
   AND L1.FROM_DATE > L2.FROM_DATE
   AND L2.TO_DATE < L1.TO_DATE
   AND L1.FROM_DATE < L2.TO_DATE
UNION
```

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID, L2.FROM_DATE, L1.TO_DATE
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
   AND L1.FDYD_ID = L2.FDYD_ID
   AND L1.PEN_ID = L2.PEN_ID
   AND L2.FROM_DATE > L1.FROM_DATE
   AND L1.TO_DATE < L2.TO_DATE
   AND L2.FROM_DATE < L1.TO_DATE
UNION
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID, L2.FROM_DATE, L2.TO_DATE
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
   AND L1.FDYD_ID = L2.FDYD_ID
   AND L1.PEN_ID = L2.PEN_ID
   AND L2.FROM_DATE >= L1.FROM_DATE
   AND L2.TO_DATE <= L1.TO_DATE
```

This query requires care to get the fourteen inequalities and the four select target lists correct. The cases where either the start times or the end times match are particularly vexing. The case where the two periods are identical (i.e., L1.FROM_DATE = L2.FROM_DATE AND L1.TO_DATE = L2.TO_DATE) is covered by two of the cases: the first and the last. This introduces an undesired duplicate. However, the UNION operator automatically removes duplicates, so the result is correct.

The downside of using UNION is that it does a lot of work to remove these infrequent duplicates generated during the evaluation of the join. We can replace UNION with UNION ALL which retains duplicates, and generally runs faster. If we do that, then we must also add the following to the predicate of the last case.

```
AND NOT (L1.FROM_DATE = L2.FROM_DATE AND L1.TO_DATE = L2.TO_DATE)
```

The result of this query contains two rows.

LOT_ID_NUM	LOT_ID_NUM	PEN_ID	FROM_DATE	TO_DATE
219	374	1	1998-02-25	1998-03-01
219	374	1	1998-03-01	1998-03-14

This result contains no sequenced duplicates (at no time are there two rows with the same values for the non-timestamp columns). Converting this result into the equivalent, but shorter,

LOT_ID_NUM	LOT_ID_NUM	PEN_ID	FROM_DATE	TO_DATE
219	374	1	1998-02-25	1998-03-14

is a story unto itself.

The SQL-92 CASE expression allows this query to be written as a single SELECT statement.

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID,
      CASE WHEN L1.FROM_DATE > L2.FROM_DATE
         THEN L1.FROM_DATE
         ELSE L2.FROM_DATE END,
      CASE WHEN L1.TO_DATE > L2.TO_DATE
         THEN L2.TO_DATE
         ELSE L1.TO_DATE END
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
   AND L1.FDYD_ID = L2.FDYD_ID
   AND L1.PEN_ID = L2.PEN_ID
   AND (CASE WHEN L1.FROM_DATE > L2.FROM_DATE
         THEN L1.FROM_DATE
```

```
    ELSE L2.FROM_DATE END) <
  (CASE WHEN L1.TO_DATE > L2.TO_DATE
    THEN L2.TO_DATE
    ELSE L1.TO_DATE END)
```

The first CASE expression simulates a last_instant function of two arguments, the second a first_instant function of the two arguments. The additional where predicate ensures the period of validity is well formed, that its starting instant occurs before its ending instant. As this version is not based on UNION, it does not introduce extraneous duplicates.

2.3 DBMS Specifics

DB2 supports the CASE construct. However, other approaches are necessary to avoid the multiple UNIONs when the CASE construct is not available.

On alternative is to implement the first_instant and last_instant functions directly, as SQL/PSM (persistent stored module) FUNCTIONs.

```
CREATE FUNCTION first_instant (one DATE, two DATE)
RETURNS DATE
LANGUAGE SQL

RETURN CASE WHEN one > two
          THEN one
        ELSE two END
```

A last_instant function can similarly defined. In fact, we can exploit polymorphism in SQL/PSM by defining a host of first_instant and last_instant functions, each taking two parameters of each of the various temporal types (e.g., TIME, TIMESTAMP, TIMESTAMP(3)) and returning the same type.

With these functions, the sequenced join is considerably simplified.

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID,
      last_instant(L1.FROM_DATE, L2.FROM_DATE), first_instant(L1.TO_DATE, L2.TO_DATE)
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
  AND L1.FDYD_ID = L2.FDYD_ID
  AND L1.PEN_ID = L2.PEN_ID
  AND last_instant(L1.FROM_DATE, L2.FROM_DATE) < first_instant(L1.TO_DATE, L2.TO_DATE)
```

Unfortunately, vendors don't (yet) support the SQL/PSM standard. In Informix-SQL, the first_instant function can be written as a PROCEDURE that returns a result.

```
CREATE PROCEDURE first_instant (one DATE, two DATE)
  RETURNING DATE;
  IF one < two THEN RETURN one;
  ELSE RETURN two;
  END IF
END PROCEDURE;
```

Several vendors also support user-defined functions, generally in the C language, for example Informix's Universal Server and IBM's DB2.

While Oracle does not support SQL-92's CASE expression, it does support the functions GREATEST and LEAST, which are generalizations of the last_instant and first_instant SQL/PSM functions defined above. (In the following, we use TODATE, because TO_DATE is a predefined function in Oracle.)

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID,
      GREATEST(L1.FROM_DATE, L2.FROM_DATE), LEAST(L1.TODATE, L2.TODATE)
FROM LOT_LOC AS L1, LOT_LOC AS L2
```

```
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
   AND L1.FDYD_ID = L2.FDYD_ID
   AND L1.PEN_ID = L2.PEN_ID
   AND GREATEST(L1.FROM_DATE, L2.FROM_DATE) < LEAST(L1.TODATE, L2.TODATE)
```

In summary, we have investigated current, nonsequenced, and sequenced variants of common types of queries. Current queries are easy: add a currency predicate for each correlation name in the from clause. Non-sequenced variants are also straight-forward: just ignore the timestamp columns, or treat them as regular columns.

Sequenced queries are of the form, "Give the history of..." and arise frequently. For projections, selections, union, and order by, of which only the first two are exemplified here, the conversion is also easy: just append the timestamp columns to the target list of the select statement. Sequenced temporal joins, however, can be awkward unless a CASE construct or first_instant type of function is available.

In the section, we'll tackle modification of valid-time state tables.

All the above approaches assume that the underlying table contains no sequenced duplicates. As a challenge, consider performing in SQL a temporal join on a table possibly containing such duplicates. The result should respect the duplicates of the input table. If that is too easy, try writing in SQL the sequenced query, "Give the history of the number of cattle in pen 1." This would return the following.

PEN_ID	HD_CNT	FROM_DATE	TO_DATE
1	17	1998-02-07	1998-02-18
1	14	1998-02-20	1998-02-25
1	57	1998-02-25	1998-03-01
1	34	1998-03-01	1998-03-14
1	14	1998-03-14	9999-12-31

3 Modifying Valid-Time State Tables

In the previous section we discussed tracking cattle as they moved from pen to pen in a feed yard. I initially hesitated in discussing this next topic due to its sensitive nature, especially for the animals concerned. But the epidemiologic factors convinced me to proceed.

3.1 Terminology

An aside on terminology. A "bull" is a male bovine animal (the term also denotes a male moose). A "cow" is a female bovine animal (or a female whale). A "calf" is the young of a cow (or a young elephant). A "heifer" is a cow that has not yet borne a calf (or a young female turtle). "Cattle" are collected bovine animals.

A "steer" is a castrated male of the cattle family. To steer an automobile or a committee is emphatically different from steering a calf. Cows and heifers are not steered, they are spayed, or generically, neutered, rendering them a "neutered cow". There is no single term for neutered cow paralleling the term steer, perhaps because spaying is a more invasive surgical procedure than steering, or perhaps because those doing the naming are cowboys.

Bulls are steered to reduce injuries to themselves (bulls are quite aggressive animals) as well as to enhance meat quality. Basically, all that fighting reduces glycogen in the muscle fibers, which increases the water content of the meat, which results in less meat per pound–the water boils off during cooking. Heifers are spayed only if they will feed in open fields, because calving in the feed yard is expensive and dangerous to the cow.

Figure 3 illustrates the transitions in gender that are possible, all of which are irreversible.

Capturing the (time-varying) gender of a lot (a collection of cattle) is important in epidemiological studies, for the gender can affect disease transfer to and between cattle. Hence, Dr. Brad De Groot's feed yard database schema includes the valid-time state table LOT, a slice of which is shown in Table 2 (in this excerpt, we've omitted the FDYD_ID, IN_WEIGHT, OWNER, and several other columns not relevant to this discussion).

The GNDR_CODE is an integer code. For expository purposes, we will use single letters, with c=bull calf, h=heifer and s=steer. The FROM_DATE and TO_DATE in concert specify the time period over which the values of all the other columns of the row were valid. In this table, on March 23, 1998, a rather momentous event occurred for the cattle in lot 101: they were steered. Lot 234 consists of calves; a TO_DATE of forever denotes a row that is currently valid. Lot 234 arrived in the feed yard on February 17; lot 799 arrived on March 12.

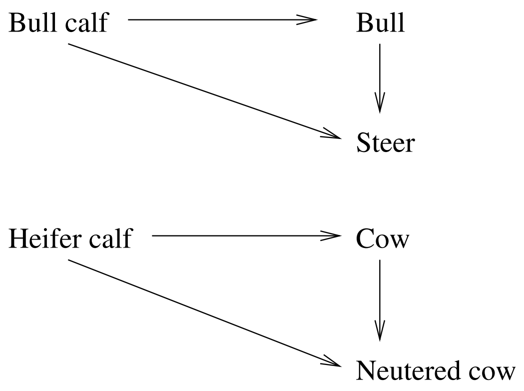


Figure 3: Gender transitions

LOT_ID_NUM	GNDR_CODE	FROM_DATE	TO_DATE
101	c	1998-01-01	1998-03-23
101	s	1998-03-23	9999-12-31
234	c	1998-02-17	9999-12-31
799	s	1998-03-12	9999-12-31

Table 2: An excerpt from the LOT table

Brad collects data from the feed yard to populate his database. In doing so he makes a series of modifications to his tables, including the LOT table (modifications comprise insertions, deletions, and updates). We previously presented current, sequenced, and non-sequenced uniqueness constraints and queries. So you've probably already guessed that I'll be discussing here current, sequenced and nonsequenced modifications.

3.2 Current Modifications

Consider a new lot of heifers that arrives today. The current insertion would be coded in SQL as follows.

```
INSERT INTO LOT
VALUES (433, 'h', CURRENT_DATE, DATE '9999-12-31')
```

The statement provides a timestamp from "now" to the end of time.

The message from previous case studies is that it is best to initially ignore the timestamp columns, as they generally confound rather than illuminate. Consider lot 101 leaving the feed yard. Ignoring time, this would be expressed as a deletion.

```
DELETE FROM LOT
WHERE LOT_ID_NUM = 101
```

A logical current deletion on a valid-time state table is expressed in SQL as an update. Current deletions apply from "now" to "forever".

```
UPDATE LOT
SET TO_DATE = CURRENT_DATE
WHERE LOT_ID_NUM = 101
  AND TO_DATE = DATE '9999-12-31'
```

There are two scenarios to consider: the general scenario, where any modification is allowed to the valid-time state table, and the restricted scenario, where only current modifications are performed on the table. The scenarios

LOT_ID_NUM	GNDR_CODE	FROM_DATE	TO_DATE
101	c	1998-01-01	1998-03-23
101	s	1998-03-23	9999-12-31
234	c	1998-02-17	1998-10-17
234	s	1998-10-17	9999-12-31
799	c	1998-03-12	9999-12-31

Table 3: Another excerpt, the general scenario

LOT_ID_NUM	GNDR_CODE	FROM_DATE	TO_DATE
101	c	1998-01-01	1998-03-23
101	s	1998-03-23	9999-12-31
234	c	1998-02-17	1998-07-29
799	c	1998-03-12	9999-12-31

Table 4: Result of a current deletion on Table 3

differentiate the data upon which the modification is performed, and consider whether a non-current modification might have been performed in the past. Often we know a priori that only current modifications are possible, which tells us something about the data that we can exploit in the (current) modification being performed.

The above statement works only in the restricted scenario. Consider the excerpt of LOT shown in Table 3, which is the general scenario. Assume today is July 29. This table indicates that lot 234 is scheduled to be steered on October 17, though we don't tell that to the calves...

A logical current deletion of lot 234, meaning that the lot left the feed yard today, in the general scenario is implemented as a physical update and a physical delete.

```
UPDATE LOT
SET TO_DATE = CURRENT_DATE
WHERE LOT_ID_NUM = 234
  AND TO_DATE >= CURRENT_DATE
  AND FROM_DATE < CURRENT_DATE

DELETE FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE > CURRENT_DATE
```

These two statements can be done in either order, as the rows they alter are disjoint. Applying these operations to Table 3 results in Table 4. All information on lot 234 after today has been deleted.

Consider steering the cattle in lot 799. On a non-temporal table, this would be stated as

```
UPDATE LOT
SET GNDR_CODE = 's'
WHERE LOT_ID_NUM = 799
```

A logical current update is implemented as a physical delete coupled with a physical insert. So this modification on a valid-time state table in the restricted scenario is as follows. (In Oracle, one can replace the outer FROM LOT with FROM DUAL, and thereby omit DISTINCT.)

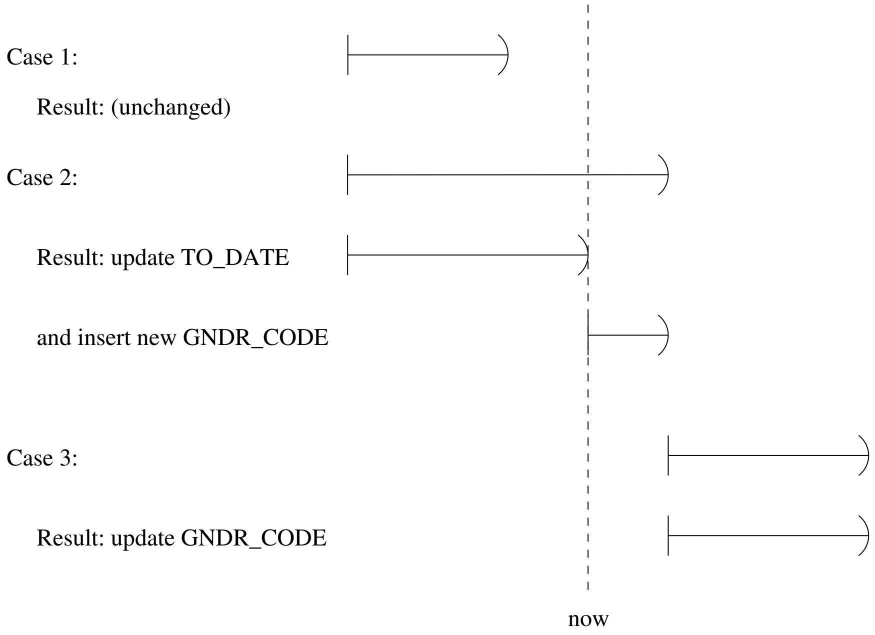


Figure 4: Current update cases, in the general scenario

```
INSERT INTO LOT
SELECT DISTINCT 799, 's', CURRENT_DATE, DATE '9999-12-31'
FROM LOT
WHERE EXISTS (SELECT *
        FROM LOT
        WHERE LOT_ID_NUM = 799
          AND TO_DATE = DATE '9999-12-31')

UPDATE LOT
SET TO_DATE = CURRENT_DATE
WHERE LOT_ID_NUM = 799
  AND GNDR_CODE <> 's'
  AND TO_DATE = DATE '9999-12-31'
```

The update terminates current values at "now" and the insert adds the new values. The update must occur after the insertion. Alternatively, the portion up to now could be inserted and the update could change the GNDR_CODE to 's' and the FROM_DATE to "now".

In the general scenario, a logical current update is more complicated, as there may exist rows that start in the future, as well as rows that end before "forever". For the former, only the GNDR_CODE need be changed. For the latter, the TO_DATE must be retained on the inserted row.

Figure 4 shows the three cases. The period of validity of the row from the table being modified is shown, with time moving left to right and "now" indicated with a vertical dashed line. In case 1, if a row's period of validity terminates in the past, then the (logical) update will not affect that row. (Recall that the logical update applies from "now" to "forever".) If the row is currently valid (case 2), then the portion before "now" must be terminated and a new row, with an updated gender, inserted, with the period of validity starting at "now" and terminating when the original row did. If the row starts in the future (case 3), the row can be updated as usual. These machinations require two updates and an insertion.

```
INSERT INTO LOT
SELECT LOT_ID_NUM, 's', CURRENT_DATE, TO_DATE
```

```
FROM LOT
WHERE LOT_ID_NUM = 799
   AND FROM_DATE <= CURRENT_DATE
   AND TO_DATE > CURRENT_DATE

UPDATE LOT
SET TO_DATE = CURRENT_DATE
WHERE LOT_ID_NUM = 799
   AND GNDR_CODE <> 's'
   AND FROM_DATE < CURRENT_DATE
   AND TO_DATE > CURRENT_DATE

UPDATE LOT
SET GNDR_CODE = 's'
WHERE LOT_ID_NUM = 799
   AND FROM_DATE >= CURRENT_DATE
```

The second update can appear anywhere, but the first update must occur after the insertion.

3.3 Sequenced Modifications

A current modification applies from "now" to "forever". A sequenced modification generalizes this to apply over a specified period, termed the period of applicability. This period could be in the past, in the future, or overlap "now".

Most of the previous discussion applies to sequenced modifications, with CURRENT_DATE replaced with the start of the period of applicability of the modification and DATE '9999-12-31' replaced with the end of the period of applicability.

In a sequenced insertion, the application provides the period of applicability. As an example, lot 426, a collection of heifers, was on the feed yard from March 26 to April 14.

```
INSERT INTO LOT
VALUES (426, 'h', DATE '1998-03-26', DATE '1998-04-14')
```

Recall that a current deletion in the general scenario is implemented as an update, for those currently valid rows, and a delete, for periods starting in the future. For a sequenced deletion, there are four cases, as shown in Figure 5. In each case, the period of validity (PV) of the original tuple is shown above the period of applicability (PA) for the deletion. In case (1), the original row covers the period of applicability, so both the initial and final periods need to be retained. The initial period is retained by setting the TO_DATE to the beginning of the period of applicability; the final period is inserted. In case (2), only the initial portion of the period of validity of the original row is retained. Symmetrically, in case (3) only the final portion of the period need be retained. And in case (4), the entire row should be deleted, as the period of applicability covers it entirely.

A sequenced deletion requires four physical modifications. We wish to record that lot 234 will be absent from the feed yard for the first three weeks of October, when the steering will take place (as recorded in Table 3). Hence, the period of applicability is DATE '1998-10-01' to DATE '1998-10-22' (we're using an TO_DATE of the day after the period ends).

```
INSERT INTO LOT
SELECT LOT_ID_NUM, GNDR_CODE, DATE '1998-10-22', TO_DATE
FROM LOT
WHERE LOT_ID_NUM = 234
   AND FROM_DATE <= DATE '1998-10-01'
   AND TO_DATE > DATE '1998-10-22'
```

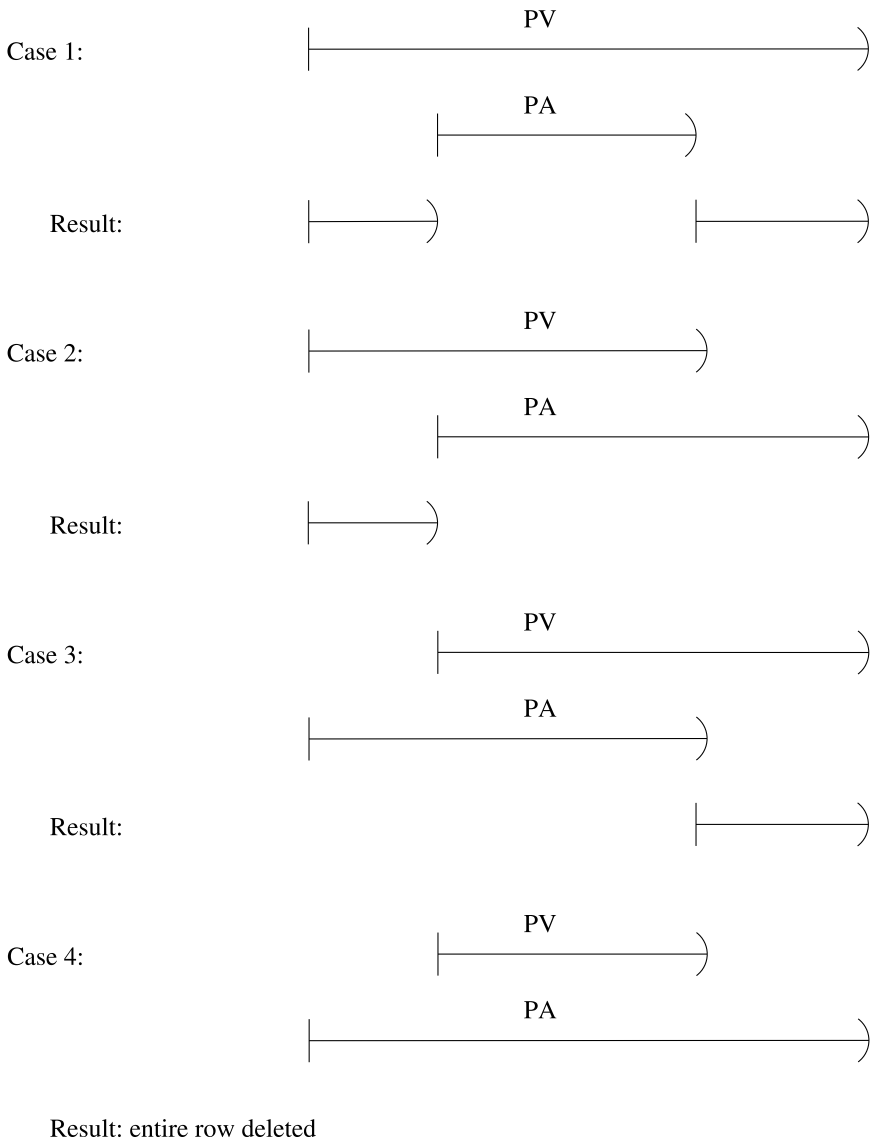


Figure 5: Sequenced deletion cases

```
UPDATE LOT
SET TO_DATE = DATE '1998-10-01'
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-01'
  AND TO_DATE >= DATE '1998-10-01'

UPDATE LOT
SET FROM_DATE = DATE '1998-10-22'
WHERE LOT_ID_NUM = 234
  AND FROM_DATE < DATE '1998-10-22'
  AND TO_DATE >= DATE '1998-10-22'

DELETE FROM LOT
WHERE LOT_ID_NUM = 234
  AND FROM_DATE >= DATE '1998-10-01'
  AND TO_DATE <= DATE '1998-10-22'
```

Case (1) is reflected in the first two statements; the second statement also covers case (2). The third statement handles case (3) and the fourth, case (4). All four statements must be evaluated in the order shown. They have been carefully designed to cover each case exactly once.

A sequenced update is the temporal analogue of a nontemporal update, with a specified period of applicability. Let us again consider steering the cattle in lot 799.

```
UPDATE LOT
SET GNDR_CODE = 's'
WHERE LOT_ID_NUM = 799
```

We now convert this to a sequenced update. As with sequenced deletions, there are more cases to consider for sequenced updates, as compared with current updates. The four cases in Figure 6 are handled differently in an update. In case 1 of the figure, the initial and final portions of the period of validity are retained (via two insertions), and the affected portion is updated. In the second case, only the initial portion is retained; in the third case, only the final portion is retained. In case 4, the period of validity is retained, as it is covered by the period of applicability.

In summary, we need to (1) insert the old values from the FROM_DATE to the beginning of the period of applicability; (2) insert the old values from the end of the period of applicability to the TO_DATE; (3) update the explicit columns of rows that overlap the period of applicability; (4) update the FROM_DATE to begin at the beginning of the period of applicability of rows that overlap the period of applicability; and (5) update the TO_DATE to end at the end of the period of applicability of rows that overlap the period of applicability.

The following is a sequenced update, recording that the lot was steered only for the month of March. (Something magical happened on April 1. The idea here is to show how to implement sequenced updates in general, and not just on cattle.) The period of applicability is thus DATE '1998-03-01' to DATE '1998-04-01'.

The first insert statement handles the initial portions of cases 1 and 2; the second handles the final portions of cases 2 and 3. The first update handles the update for all four cases. The second and third updates adjust the starting (for cases 1 and 2) and ending dates (for cases 1 and 3) of the updated portion. Note that the last three update statements will not impact the row(s) inserted by the two insert statements, as the period of validity of those rows lies outside the period of applicability. Again, all five statements must be evaluated in the order shown.

```
INSERT INTO LOT
SELECT LOT_ID_NUM, GNDR_CODE, FROM_DATE, DATE '1998-03-01'
FROM LOT
WHERE LOT_ID_NUM = 799
  AND FROM_DATE < DATE '1998-03-01'
  AND TO_DATE > DATE '1998-03-01'
```

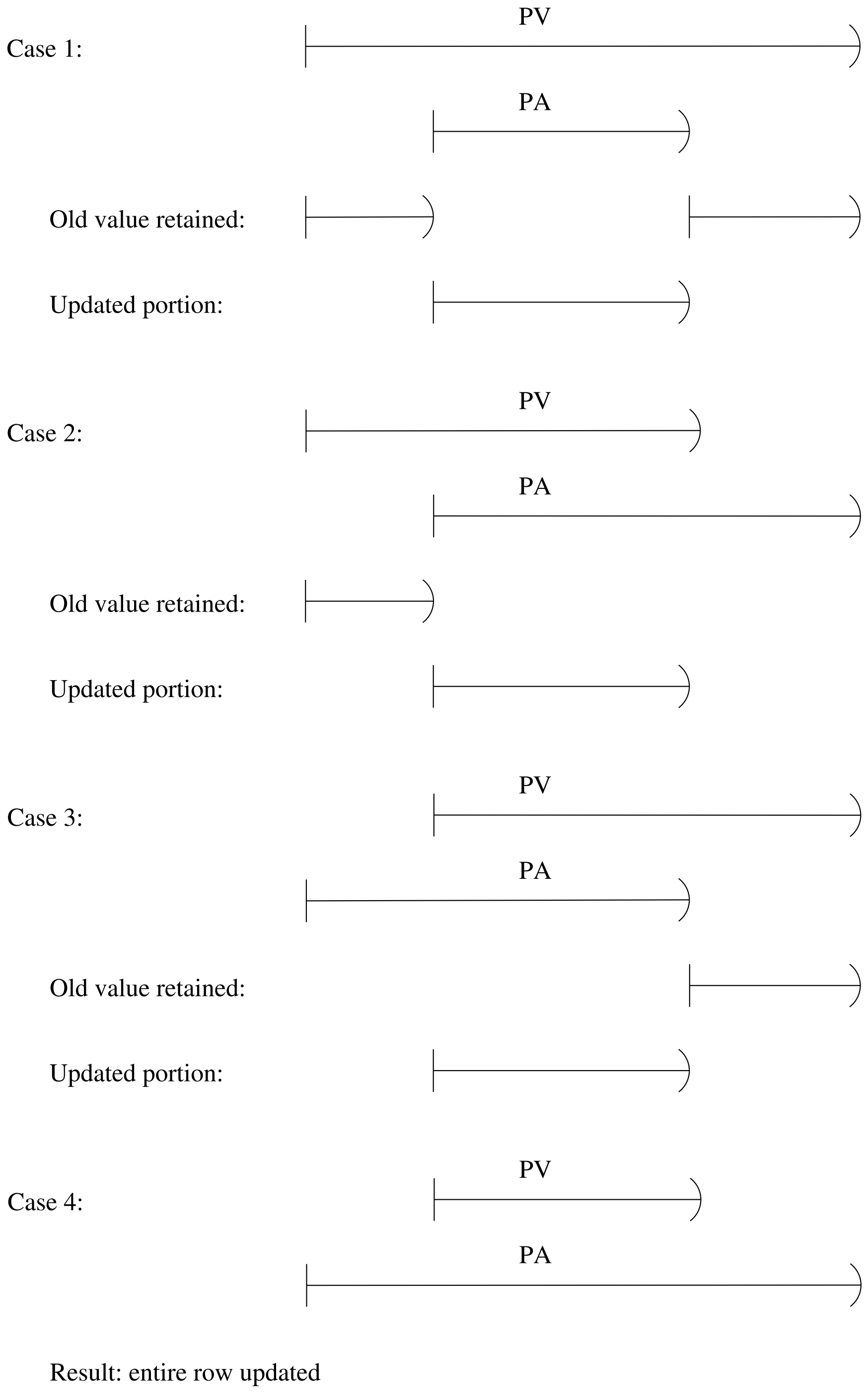


Figure 6: Sequenced update cases

```
INSERT INTO LOT
SELECT LOT_ID_NUM, GNDR_CODE, DATE '1998-04-01', TO_DATE
FROM LOT
WHERE LOT_ID_NUM = 799
  AND FROM_DATE < DATE '1998-04-01'
  AND TO_DATE > DATE '1998-04-01'

UPDATE LOT
SET GNDR_CODE = 's'
WHERE LOT_ID_NUM = 799
  AND FROM_DATE < DATE '1998-04-01'
  AND TO_DATE > DATE '1998-03-01'

UPDATE LOT
SET FROM_DATE = DATE '1998-03-01'
WHERE LOT_ID_NUM = 799
  AND FROM_DATE < DATE '1998-03-01'
  AND TO_DATE > DATE '1998-03-01'

UPDATE LOT
SET TO_DATE = DATE '1998-04-01'
WHERE LOT_ID_NUM = 799
  AND FROM_DATE < DATE '1998-04-01'
  AND TO_DATE > DATE '1998-04-01'
```

3.4 Nonsequenced Modifications

As with constraints and queries, a nonsequenced modification treats the timestamps identically to the other columns. Consider the modification, "Delete lot 234." The current variant is "Lot 234 has just left the feed yard." A sequenced variant, with a period of applicability, is "Lot 234 will be absent from the feed yard for the first three weeks of June." A nonsequenced deletion mentions the period of validity of the rows to be deleted. An example is "Delete the records of lot 234 that have duration greater than three months."

```
DELETE FROM LOT
WHERE LOT_ID_NUM = 234
  AND (TO_DATE - FROM_DATE MONTH) > INTERVAL '3' MONTH
```

The current and sequenced deletes mention what happened in reality, because they model changes. The nonsequenced statement concerns the specific representation (deleting particular records). Conversely, the associated SQL statements for the current and sequenced variants are much more complex than that for the nonsequenced delete, for the same reason: the latter is expressed in terms of the representation.

Most modifications will be first expressed as changes to the enterprise being modeled (some fact becomes true, or will be true sometime in the future; some aspect changes, now or in the future; some fact is no longer true). Such modifications are either current or sequenced modifications. Nonsequenced modifications, while generally easier to express in SQL, are rare.

For those who want a challenge, alter the above modification statements to ensure sequenced primary key and referential integrity constraints.

As a final comment, it might be surprising that a time-varying gender is relevant outside of cattle databases. I've been told that Pacific Bell's personnel database has a date field associated with gender; more than a dozen of its employees change their gender each month. Only in California...

RA_Hour	RA_Min	RA_Sec	Dec_Degree	Dec_Minute	Discoverer	Mag_First
00	00	08	75	30	'A 1248'	10.5
05	57	40	00	02	'BU 1190'	6.5
04	13	20	50	32	'CHR 15'	15.5
01	23	70	-09	55	'HJ 3433'	10.5

Table 5: An excerpt from the WDS table

4 Transaction-Time State Tables

Temporal data is data that varies over time. However, it may surprise readers that some of the approaches outlined above are applicable even when the enterprise being modeled does *not* vary over time.

Consider astronomical data, specifically, that of stars. While stars coalesce out of galactic dust, heat up, and, when their fuel is spent, explode or die out, perhaps ending up as black holes, this progression is played out over hundred of millions, or even billions, of year. For all intents and purposes, the position, magnitude (brightness) and spectral type of a star is time-invariant over a comprehensible scale such as a person's lifetime. This static nature has encouraged the compilation of star catalogues, such as the Smithsonian Astrophysical Observatory J2000 Catalogue (http://www.fc.net/ brianc/man/saoj.ms0), containing almost 300,000 stars, or the Washington Double Star Catalogue (http://aries.usno.navy.mil/ad/wds/wds.htm), containing some 78,000 double and multiple star systems.

What *is* time-varying is our knowledge about these stars. For example, the WDS is based on some 451,000 individual observations, by a host of discoverers and observers over the last century. Data is continually being incorporated, to add newly discovered binary systems and to refine the data on known systems, some of which enjoy as many as 100 individual observations.

The challenge in assembling such a catalogue lies in correlating the data and winnowing out inconsistent or spurious measurements. As such, it is desirable to capture with each change to the catalogue when that change was made, as well as additional information such as who made the change and the source of the new information. In this way, past versions of the catalogue can be reconstructed, and the updates audited, to enable analysis of both the resulting catalogue and of its evolution.

We previously considered valid-time state tables, which model time-varying behavior of an enterprise. We now examine *transaction-time state tables*, which record an evolving understanding of some static system.

A subtle but critical paradigm shift is at play here. A valid-time table models the fluid and continual movement of reality: cattle are transfered from pen to pen, a caterpillar becomes a chrysalis in its cocoon, later to emerge as a butterfly, salaries rise (and sometimes fall) in fits and sputters. A transaction-time table instead captures the succession of states of the stored representation of some (static) fact: a star was thought to have a particular spectral type but is later determined to have somewhat different spectral characteristics, the bond angle within a chemical structure is refined as new X-ray diffraction data becomes available, intermediate configurations within a nuclear transformation are corrected as accelerator data is analyzed.

These two characterizations of time-varying behavior, valid time and transaction time, are orthogonal. We will consider for the most part only transaction time here, bringing it together with valid time in one gloriously expressive structure only at the end.

4.1 Transaction-Time State Tables

We consider a subset of the Washington Double Star (WDS) catalogue. The WDS Bible contains 21 columns; only a few will be used here.

RA denotes "Right Ascension"; Dec denotes "declination"; these first five columns place the star positionally in the heavens. The discoverer is identified by a one-to-three letter code, along with a discoverer's number. This column provides the primary key for the table. The last column records the magnitude (brightness) or the first component of the dual or multiple star system.

RA_ Hour	RA_ Min	RA_ Sec	Dec_ Degree	Dec_ Minute	Discoverer	Mag_ First	Trans_ Start	Trans_ Stop
00	00	00	75	30	'A 1248'	12.0	1989-03-12	1992-11-15
00	00	09	75	30	'A 1248'	12.0	1992-11-15	1994-05-18
00	00	09	75	30	'A 1248'	10.5	1994-05-18	1995-07-23
00	00	08	75	30	'A 1248'	10.5	1995-07-23	9999-12-31
05	57	40	00	02	'BU 1190'	6.5	1988-11-08	9999-12-31
04	13	20	50	32	'CHR 15'	15.5	1990-02-09	9999-12-31
01	23	70	-09	55	'HJ 3433'	10.5	1991-03-25	9999-12-31
02	33	10	-09	25	'LDS3402'	10.6	1993-12-19	1996-07-09

Table 6: The audit log (WDS_TT) for the WDS table

As mentioned previously, this table is constantly updated with new binary stars and with corrections to existing stars. To track these changes, we define a new table, WDS_TT, with two additional columns, Trans_Start and Trans_Stop, yielding a transaction-time state table. We term this table an *audit log*, differentiating it from the original table, which has no timestamps. Trans_Start specifies when the row was inserted into the original table, or when the row was updated (the new contents of the row are recorded here). Trans_Stop specifies when the row was deleted from the original table, or was updated (the old contents of the row are recorded here). Consider the following audit log for the WDS table. We show the timestamps as DATEs, but they often are of much finer granularity, such as TIMESTAMP(6), to distinguish multiple transactions occurring in a day, or even within a single second.

A Trans_Stop time of "forever" (9999-12-31) indicates that the row is currently in WDS. And as we saw above, WDS currently contains four rows, so four rows of WDS_TT have a Trans_Stop value of "forever". The binary star LDS3402 was inserted the end of 1993, then deleted in July, 1996, when it was found to be in error. The binary star A 1248 was first inserted in 1989, and was subsequently modified, in November 1992 (to correct its RA_Sec position), May 1994 (to refine its magnitude), and July 1995 (to refine its position slightly). Note that these changes do not mean that the *star* is changing, rather that the prior measurements were in error, and have since been corrected. Rows with a past Trans_Stop date are (now) known to be incorrect.

4.2 Maintaining The Audit Log

The audit log can be maintained automatically using triggers defined on the original table. The advantage to doing so is that the applications that maintain the WDS table need not be altered at all when the audit log is defined. Instead, the audit log is maintained purely as a side-effect of the modifications applied to the original table.

Using triggers has another advantage: it simplifies specifying the primary key of the audit log. In Section 1, we saw that it is challenging to define unique columns or a primary key for a valid-time state table. Not so for a transaction-time state table: all that is necessary is appending Trans_Start to the primary key of the original table. Hence, the primary key of WDS_TT is (Discoverer, Trans_Start).

The triggers ensure that the audit log captures all the changes made to the original table. When a row is inserted into the original table, it is also inserted into the audit log, with Trans_Start initialized to "now" (CURRENT_DATE), and Trans_Stop is initialized to "forever". To logically delete a row, the Trans_Stop of the row is changed to "now" in the audit log. An update is handled as a deletion followed by an insertion.

```
CREATE TRIGGER INSERT_WDS
AFTER INSERT ON WDS
REFERENCING NEW AS N
FOR EACH ROW
INSERT INTO WDS_TT(RA_Hour, RA_Min, RA_Sec, Dec_Degree, Dec_Minute,
    Discoverer, Mag_First, Trans_Start, Trans_Stop)
VALUES (N.RA_Hour, N.RA_Min, N.RA_Sec, N.Dec_Degree, N.Dec_Minute,
    N.Discoverer, N.Mag_First, CURRENT_DATE, DATE '9999-12-31')
```

```
CREATE TRIGGER DELETE_WDS
AFTER DELETE ON WDS
REFERENCING OLD AS O
FOR EACH ROW
UPDATE WDS_TT
SET STOP_TIME = CURRENT_DATE
WHERE WDS_TT.Discoverer = O.Discoverer
  AND WDS_TT.Trans_Stop = DATE '9999-12-31'

CREATE TRIGGER UPDATE_P
AFTER UPDATE ON WDS
REFERENCING OLD AS O NEW AS N
FOR EACH ROW
BEGIN ATOMIC
  UPDATE WDS_TT
  SET Trans_Stop = CURRENT_DATE
  WHERE WDS_TT.Discoverer = O.Discoverer
    AND WDS_TT.Trans_Stop = DATE '9999-12-31';
  INSERT INTO WDS_TT(RA_Hour, RA_Min, RA_Sec,
        Dec_Degree, Dec_Minute, Discoverer, Mag_First,
        Trans_Start, Trans_Stop)
  VALUES (N.RA_Hour, N.RA_Min, N.RA_Sec,
        N.Dec_Degree, N.Dec_Minute, N.Discoverer, N.Mag_First,
        CURRENT_DATE, DATE '9999-12-31');
END
```

These triggers could be augmented to also store other information in the audit log, such as CURRENT_USER.

Note that WDS_TT is monotonically increasing in size. The INSERT trigger adds a row to WDS_TT, the DELETE trigger just changes the value of the Trans_Stop column, and the UPDATE trigger does both, adding one row and updating another. No row is ever deleted from WDS_TT.

4.3 Querying The Audit Log

In Section 2, we discussed three variants of queries on valid-time state tables: current, sequenced, and nonsequenced. These variants also apply to transaction-time state tables. To determine the current state of the WDS table, we can either look directly to that table, or get the information from the audit log.

```
SELECT RA_Hour, RA_Min, RA_Sec, Dec_Degree, Dec_Minute, Discoverer, Mag_First
FROM WDS_TT
WHERE Trans_Stop = DATE '9999-12-31'
```

The utility of an audit log becomes apparent when we wish to *rollback* the WDS table to its state as of a previous point in time. Say we wish to see the WDS table as it existed on April 1, 1994. This reconstruction is best expressed as a view.

```
CREATE VIEW WDS_April_1 AS
SELECT RA_Hour, RA_Min, RA_Sec, Dec_Degree, Dec_Minute, Discoverer, Mag_First
FROM WDS_TT
WHERE Trans_Start <= DATE '1994-04-01' AND DATE '1994-04-01' < Trans_Stop
```

The result of this is Table 7.

Note that LDS3402 is present here (the mistake hadn't yet been detected), and that A1248 has an incorrect magnitude and position (these errors also hadn't been corrected as of April 1, 1994). What we've done here is rolled back time to April 1, 1994, to see what the WDS table looked like at that time. Queries on WDS_April_1 will return the same result as queries on WDS that were presented to the DBMS on that date. So, if we ask, which stars are of magnitude 11 or brighter, as currently known,

RA_ Hour	RA_ Min	RA_ Sec	Dec_ Degree	Dec_ Minute	Discoverer	Mag_ First
00	00	09	75	30	'A 1248'	12.0
05	57	40	00	02	'BU 1190'	6.5
04	13	20	50	32	'CHR 15'	15.5
01	23	70	-09	55	'HJ 3433'	10.5
02	33	10	-09	25	'LDS3402'	10.6

Table 7: Transaction timeslice of WDS_T as of April 1, 1994.

```
SELECT Discoverer
FROM WDS
WHERE Mag_First <= 11.0
```

(brighter stars have smaller magnitudes), three double stars would be identified.

Discoverer
'A 1248'
'BU 1190'
'HJ 3433'

Asking the same question, as best known on April 1, 1994,

```
SELECT Discoverer
FROM WDS_April_1
WHERE Mag_First <= 11.0
```

yields a different set of stars,

Discoverer
'BU 1190'
'HJ 3433'
'LDS3402'

because A1248 was thought then (erroneously) to be of magnitude 12, and LDS3402 was thought then (also erroneously) to be a double star system, of magnitude 10.6.

Interestingly, the WDS_April_1 can also be defined as a table, instead of as a view. The reason is that no future modifications to the WDS table will alter the state of that table back in April, and so any future query of WDS_April_1, whether a view or a table, will return the same result, independently of when that query is specified. The decision to make WDS_April_1 a view or a table is entirely one of query efficiency versus disk space.

We emphasize that only past states can be so queried. Even though the Trans_Stop value is "forever" (chosen to make the queries discussed below easier to write), this must be interpreted as "now". We cannot unequivocally state what the WDS table will record in the future; all we know is what is recorded now in that table, and the (erroneous) values that were previously recorded in that table.

Sequenced and nonsequenced queries are also possible on transaction-time state tables. Consider the query, "When was it recorded that A1248 had a magnitude other than 10.5?" The first part, "when was it recorded" indicates that we are concerned with transaction time, and thus must use the WDS_TT table. It also implies that if a particular time is returned, the specified relationship should hold during that time. This indicates a sequenced query. In this case, the query is a simple selection and projection.

```
SELECT Mag_First, Trans_Start, Trans_Stop
FROM WDS_TT
WHERE Discoverer = 'A 1248'
  AND Mag_First <> 10.5
```

The following result

Mag_ First	Trans_ Start	Trans_ Stop
12.0	1989-03-12	1992-11-15
12.0	1992-11-15	1994-05-18

indicates that for a little over 5 years, the magnitude of the first star in this double star system was recorded incorrectly in the database.

We can use all the tricks discussed in Section 3 to write sequenced queries on WDS_TT. The query "When was it recorded that a star had a magnitude equal to that of A1248?" The first part again indicates a transaction-time sequenced query; the last part indicates a self-join. This can be expressed in Oracle as

```
SELECT W1.Discoverer,
       GREATEST(W1.Trans_Start, W2.Trans_Start), LEAST(W1.Trans_Stop, W2.Trans_Stop)
FROM WDS_TT W1, WDS_TT W2
WHERE W1.Discoverer = 'A 1248'
  AND W2.Discoverer <> W1.Discoverer
  AND W1.Mag_First = W2.Mag_First
  AND GREATEST(W1.Trans_Start, W2.Trans_Start) < LEAST(W1.Trans_Stop, W2.Trans_Stop)
```

This results in

Discoverer	Trans_ Start	Trans_ Stop
'HJ 3433'	1994-05-18	1995-07-23
'HJ 3433'	1995-07-23	9999-12-31

stating that in May 1994 it was recorded that HJ3433 had the same magnitude as A1248, and this is still thought to be the case.

Nonsequenced queries on transaction-time tables are effective in identifying changes. "When was the RA_Sec position of a double star corrected?" A correction is indicated by two rows that meet in transaction time, and that concern the same double star, but have different RA_Sec values.

```
SELECT W1.Discoverer, W1.RA_Sec AS Old_Value, W2.RA_Sec AS New_Value,
         W1.Trans_Stop AS When_Changed
FROM WDS_TT AS W1, WDS_TT AS W2
WHERE W1.Discoverer = W2.Discoverer
  AND W1.Trans_Stop = W2.Trans_Start
  AND W1.RA_Sec <> W2.RA_Sec
```

The result indicates that the position of A1248 was changed twice, first from 0 to 9, and then to 8.

Discoverer	Old_ Value	New_ Value	When_ Changed
'A 1248'	00	09	1992-11-15
'A 1248'	09	08	1995-07-23

4.4 Modifying The Audit Log

While queries on transaction-time tables can be current, sequenced, or non-sequenced, the same does *not* hold for modifications. In fact, the audit log (WDS_TT) should be changed only as a side effect of modifications on the original table (WDS). In the terminology introduced in Section 3 on valid-time state table modifications, the only modifications possible on transaction-time state tables are current modifications, effecting the currently stored state. The triggers defined above are very similar to the current modifications described for valid-time tables.

Sequenced and non-sequenced modifications can change the previous state of a valid-time table. But doing so to an audit log violates the semantics of that table. Say we manually insert today into WDS_TT a row with a Trans_Start value of 1994-04-01. This implies that the WDS table on that date also contained that same row. But we can't change the past, specifically, what bits were stored on the magnetic disk. For this reason, manual changes to an audit log should not be permitted; only the triggers should modify the audit log.

Discoverer	Mag_First	Trans_Start	Trans_Stop	Valid_From	Valid_To
'A 1248'	12.0	1989-03-12	1995-11-15	1922-05-14	9999-12-31
'A 1248'	12.0	1995-11-15	9999-12-31	1922-05-14	1994-10-16
'A 1248'	10.5	1995-11-15	9999-12-31	1994-10-16	9999-12-31

Table 8: A bitemporal table (WDS_B)

4.5 Bitemporal Tables

Because valid time and transaction time are orthogonal, it is possible for each to be present or absent independently. When both are supported simultaneously, the table is called a *bitemporal* table.

While stars are stationary to the eye, sophisticated astronomical instruments can sometimes detect slight motion of some stars. This movement is called "proper motion", to differentiate it from the apparent movement of the stars in the night-time sky as the earth spins. Star catalogues thus list the star's position as of a particular "epoch", or point in time. The Washington Double Star catalogue lists each star system's location as of January 1, 2000, the so-called J2000 epoch. It also indicates the proper motion, in units of seconds of arc per 1000 years. Some star systems are essentially stationary; BU733 is highly unusual in that it moves almost an arc second a year, both in ascension and in declination. Stars can sometimes also change magnitude.

We can capture this information in a bitemporal table, WDS_B. Here we show how this table might look.

This table has two transaction timestamps, and thus records transaction states (the period of time a fact was recorded in the database). The table also has two valid-time timestamps, and thus records valid-time states (the period of time when something was true in reality). While the transaction timestamps should generally be of a finer granularity (e.g., microseconds), the valid time is often much coarser (e.g., day).

Bitemporal tables are initially somewhat challenging to interpret, but such tables can express complex behavior quite naturally. The first photographic plate containing A1248 (presumably by discoverer A, who is R.G. Aitken, who was active in double star sitings for the first four decades of the twentieth century) was taken on May 14, 1922. However, this information had to wait almost 70 years before being entered into the database, in March 1989. This row has a Valid_To date of "forever", meaning that the magnitude was not expected to change. A subsequent plate was taken in October 1994, indicating a slightly brighter magnitude (perhaps the star was transitioning to a supernova), but was not entered into the database until November 1995. This logical update was recorded in the bitemporal table by updating the Trans_Stop date for the first row to "now", and by inserting two more rows, one indicating that the magnitude of 12 was only for a period of years following June 1922, and indicating that a magnitude of 10.5 was valid after 1994. (Actually, we don't know exactly when the magnitude changed, only that it had changed by the time the October 1994 plate was taken. In other applications, the valid-time from and to dates are generally quite accurately known.)

Modifications to a bitemporal table can specify the valid time, be of any of the varieties, current, sequenced, or non-sequenced. However, the transaction time must always be taken from CURRENT_DATE, or better, CURRENT_TIMESTAMP, when the modification was being applied.

Queries can be current, sequenced, or non-sequenced, for both valid and transaction time, in any combination. As one example, consider "What was the history recorded as of January 1, 1994?" "History" implies sequenced in valid time; "recorded as" indicates a transaction timeslice.

```
CREATE VIEW WDS_VT_AS_OF_Jan_1 AS
SELECT Discoverer, Mag_First, Valid_From, Valid_To
FROM WDS_B
WHERE Trans_Start <= DATE '1994-01-01'
  AND DATE '1994-01-01' < Trans_Stop
```

This returns a valid-time state view, in this case, just the first row of the above table. Valid-time queries can then be applied to this view. This effectively rolls back the database to the state stored on January 1, 1994; valid-time queries on this view will return exactly the same result as valid-time queries actually typed in on that date.

Now consider "List the corrections made on plates taken in the 1920's." "corrections" implies non-sequenced in transaction time; "taken in the 1920's" indicates sequenced in valid time. This query can be expressed in Oracle as

```
SELECT B1.Discoverer, B1.Trans_Stop AS When_Changed,
    GREATEST(B1.Valid_From, B2.Valid_From) AS Valid_From,
    LEAST(B1.Valid_To, B2.Valid_To) AS Valid_To
FROM WDS_B B1, WDS_B B2
WHERE B1.Discoverer = B2.Discoverer
  AND B1.Trans_Stop = B2.Trans_Start
  AND GREATEST(B1.Valid_From, B2.Valid_From) < DATE '1929-12-31'
  AND DATE '1920-01-01' < LEAST(B1.Valid_To, B2.Valid_To)
  AND GREATEST(B1.Valid_From, B2.Valid_From) < LEAST(B1.Valid_To, B2.Valid_To)
```

This query searches for pairs of rows that meet in transaction time, that were valid in the 1920's, and that overlap in valid time. For the above data, one such change is identified.

Discoverer	When_ Changed	Valid_ From	Valid_ To
'A 1248'	1995-11-15	1922-05-14	1994-10-16

This result indicates that erroneous data concerning information during the period from 1922 to 1994 was corrected in the database in November 1995.

Bitemporal tables record the history of the modeled reality, as well as recording when that history was stored in the database, perhaps erroneously. They are highly useful when the application needs to know both when some fact was true, *and* when that fact was known, i.e., stored in the database.

5 Temporal Support in Standard SQL

The previous four sections have shown that expressing integrity constraints, queries, and modifications on time-varying data in SQL is challenging. This final section looks to the future, examining enhancements to SQL that bring temporal processing to the masses. With just a few additional concepts, SQL can just as easily express temporal queries as it does now for nontemporal queries.

Many knotty problems arise when we have to contend with time-varying data in SQL.

- Avoiding duplicates in a time-varying table requires an aggregate or complex trigger.
- A simple three-line join when applied to time-varying tables explodes to a 37-line query consisting of four SELECT statements or a complex 20-line statement with four CASE expressions.
- A three-line UPDATE of a time-varying table translates into five modification statements totaling 29 lines.
- Maintaining an audit log requires several triggers comprising some three dozen lines.

What is the source of this daunting complexity? While SQL-92 supports time-varying data through the DATE, TIME, and TIMESTAMP data types, the language really has no notion of a time-varying table. SQL also has no concept of current or sequenced constraints, queries, modifications or views, nor of the critical distinction between valid time (modeling the behavior of the enterprise in reality) and transaction time (capturing the evolution of the stored data). In the terminology introduced before, all that SQL supports is nonsequenced operations, which we saw were often the least useful.

Fortunately, there are now specific proposals for temporal support in SQL3 that are being considered by the standards committees (see Section 5.4) and are starting to be incorporated into products by vendors. Here I will summarize these new SQL3 constructs and revisit the applications discussed above, showing how these constructs greatly simplify writing SQL for time-varying applications. In the following, when I mention an SQL3 construct, I am referring to the constructs introduced in the proposals referenced in Section 5.4, or are already present in the draft SQL3. I should emphasize that these proposals are still under consideration for SQL3. The constructs may well change; indeed, SQL3 as a whole is still undergoing refinement as it inches towards publication as an international standard.

5.1 SQL

SQL3 adds a new data type, PERIOD, actually, a series of data types: PERIOD(DATE), of a day granularity, PERIOD(TIME) of a second granularity, and PERIOD(TIMESTAMP) of a microsecond granularity, with additional variants possible by specifying a precision, e.g., PERIOD(TIMESTAMP(3)) has a granularity of 10^{-3} seconds (milliseconds) and a range of 9999 years.

SQL3 also adds the notion of a *table with temporal support*. The table can have valid-time support, transaction-time support, or both (bitemporal support). Finally, SQL3 provides facilities for current and sequenced operations, and retains the ability to perform nonsequenced operations.

Section 1 considered a table called NICUStatus (Neonatal Intensive Care Unit Status), and a uniqueness integrity constraint. Let's assume that the application was initially non-temporal, in that it captured in the NICUStatus table only the current situation: the infants present in the NICU and their current status.

```
CREATE TABLE NICUStatus (
  Name CHAR(15),
  Status CHAR(8),
  UNIQUE (Name, Status)
)
```

We now wish to retain the history of this information: how the status of infants varied over time. In SQL3, this can be done with an ALTER statement.

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

Here we add valid-time support, at the granularity of a day. Each row of the table is now associated with a valid-time period. The rows in the table when valid-time support was added are associated with the period from "now" to "forever".

SQL3 is *temporally upward compatible*, meaning that the non-temporal application is unaffected when temporal support is added. The benefit of this important property is that the tens of thousands of lines of code associated with the NICU application continue to work as before, without altering a single line of code. Queries in this application code on this table which now has valid-time support are interpreted as current queries. To ascertain which infants are currently in serious condition, the following query

```
SELECT Name
FROM NICUStatus
WHERE Status = 'serious'
```

works just as before.

Modifications are interpreted as current modifications, capturing the history. To update Alexis May's status to fair, the following modification

```
UPDATE NICUStatus
SET Status = 'fair'
WHERE Name = 'Alexis May'
```

still works fine, and also automatically retains the prior status, as a side effect of the table having valid-time support.

Integrity constraints are handled in the same way. The initial uniqueness constraint, specified when the table was created, is considered to be a current constraint: no infant can (currently) have two identical status values.

Expressing the sequenced analogue, at no time can an infant have two identical conditions, requires an aggregate or trigger in SQL-92. In SQL3, sequenced statements are indicated with the keyword VALIDTIME:

```
ALTER TABLE NICUStatus ADD CONSTRAINT VALIDTIME UNIQUE (Name, Status)
```

One line in SQL3 suffices for many in SQL-92. We will encounter similarly dramatic reductions in code size again and again as we express our application in SQL3.

5.2 Back in the Pens

In Section 2, we looked at recording the movement of cattle between pens. Here too a valid-time table is appropriate.

```
CREATE TABLE LOT_LOC (
   FDYD_ID INT,
   LOT_ID_NUM INT,
   PEN_ID INT,
   HD_CNT INT
) AS VALIDTIME PERIOD(DATE)
```

As before, current queries require no special attention. “How many head of cattle from lot 219 in feed yard 1 are (currently) in each pen?”

```
SELECT PEN_ID, HD_CNT
FROM LOT_LOC
WHERE FDYD_ID = 1 AND LOT_ID_NUM = 219
```

The sequenced variant, “Give the history of the number of head of cattle from lot 219 in feed yard 1 in each pen,” is trivial: just prepend VALIDTIME:

```
VALIDTIME SELECT PEN_ID, HD_CNT
FROM LOT_LOC
WHERE FDYD_ID = 1 AND LOT_ID_NUM = 219
```

The nonsequenced variant, “How many head of cattle from lot 219 in yard 1 were, at some time, in each pen?”, is also easy to specify: just prepend NONSEQUENCED VALIDTIME:

```
NONSEQUENCED VALIDTIME SELECT PEN_ID, HD_CNT
FROM LOT_LOC
WHERE FDYD_ID = 1 AND LOT_ID_NUM = 219
```

These two rules (prepend VALIDTIME for sequenced, and NONSEQUENCED VALIDTIME for nonsequenced) applies to *all* nontemporal SQL statements. Consider joins. The current join, “Which lots are co-resident in a pen?” could be written by anyone knowing SQL in about two minutes.

```
SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
   AND L1.FDYD_ID = L2.FDYD_ID
   AND L1.PEN_ID = L2.PEN_ID
```

The sequenced version, “Give the history of...”, in SQL3 requires but a few more seconds to write: add one keyword.

```
VALIDTIME SELECT L1.LOT_ID_NUM, L2.LOT_ID_NUM, L1.PEN_ID
FROM LOT_LOC AS L1, LOT_LOC AS L2
WHERE L1.LOT_ID_NUM < L2.LOT_ID_NUM
   AND L1.FDYD_ID = L2.FDYD_ID
   AND L1.PEN_ID = L2.PEN_ID
```

Recall that this query in SQL-92 is 37 lines long! (See Section 2.)

The nonsequenced version, “Which lots were in the same pen, perhaps at different times?” requires adding NONSEQUENCED.

Section 3 considered modifications applied to the following table, with valid-time support, which captures the (changing) gender of lots of cattle.

```
CREATE TABLE LOT (
   LOT_ID_NUM INT,
   GNDR_CODE CHAR(1)
) AS VALIDTIME PERIOD(DATE)
```

Current modifications on such tables require no additional keywords.

"A new lot of heifers arrives today."

```
INSERT INTO LOT
VALUES (433, 'h')
```

"Lot 101 is leaving the feed yard."

```
DELETE FROM LOT
WHERE LOT_ID_NUM = 101
```

"The cattle in lot 799 are being steered today."

```
UPDATE LOT
SET GNDR_CODE = 's'
WHERE LOT_ID_NUM = 799
```

This simple modification when expressed in SQL-92 requires an INSERT and two UPDATEs, or 16 lines of SQL-92 code.

By now, the reader may guess (correctly!) that a sequenced modification is signaled with the keyword VALIDTIME. The period of applicability is assumed to be over all time; it can also be explicitly stated with a period expression following this keyword.

"Lot 426, a collection of heifers, was on the feed yard from March 26 to April 14."

```
VALIDTIME PERIOD '[1998-03-26 - 1998-04-14)'
INSERT INTO LOT
VALUES (426, 'h')
```

Here we see the first use of a period literal. The left '[' bracket indicates the period includes March 26; the right ')' specifies that the period ends just before (or during) April 14.

Sequenced modifications can apply in the past, in the future, or in the past through the future.

"Lot 234 will be absent from the feed yard for the first three weeks of October, when the steering will take place."

```
VALIDTIME PERIOD '[1998-10-01 - 1998-10-22)'
DELETE FROM LOT
WHERE LOT_ID_NUM = 234
```

Consider "The lot was steered only for the month of March," which, while difficult to effect, does illustrate a sequenced update.

```
VALIDTIME PERIOD '[1998-03-01 - 1998-04-01)'
UPDATE LOT
SET GNDR_CODE = 's'
WHERE LOT_ID_NUM = 799
```

This update when expressed in SQL-92 requires two INSERT statements and three UPDATE statements, or some 29 lines of code.

Section 4 considered transaction time, specifically, capturing the succession of states of the Washington Double Star Catalogue.

```
CREATE TABLE WDS_TT (
   RA_Hour INT,
   RA_Min INT,
   RA_Sec INT,
   Dec_Degree INT,
   Dec_Minute INT,
   Discoverer CHAR(7),
   Mag_First DECIMAL(5,2)
) AS TRANSACTIONTIME
```

Here the precision is automatically assigned by the DBMS, but is sufficient to distinguish transactions (and so is probably on the order of microseconds, if the DBMS can sustain a high transaction processing rate).

Recall that maintaining this audit log in SQL-92 required three onerous triggers, or 37 lines of code; SQL3 requires but two additional keywords. More importantly, SQL3 will ensure that the semantics of transaction time is maintained, so that accessing prior states will obtain the correct result. It is not possible to guarantee this in SQL-92: anyone with update permission on the table could modify the audit log to reflect a different sequence of changes than had actually occurred.

As before, current queries require nothing extra on tables with temporal support. "Which stars are of magnitude 11 or brighter, as currently known?"

```
SELECT Discoverer
FROM WDS
WHERE Mag_First <= 11.0
```

Sequenced queries over all time are easy to express in SQL3. "When was it recorded that A1248 had a magnitude other than 10.5?"

```
TRANSACTIONTIME SELECT Mag_First
FROM WDS_TT
WHERE Discoverer = 'A 1248' AND Mag_First <> 10.5
```

"When was it recorded that a star had a magnitude equal to that of A1248?"

```
TRANSACTIONTIME SELECT W2.Discoverer,
FROM WDS_TT AS W1, WDS_TT AS W2
WHERE W1.Discoverer = 'A 1248'
   AND W2.Discoverer <> W1.Discoverer
   AND W1.Mag_First = W2.Mag_First
```

This query is twice as long and much more complicated in SQL-92.

Nonsequenced queries, on the other hand, are quite similar in SQL-92 and SQL3. "When was the RA_Sec position of a double star corrected?" Here, "corrected" means we look at two consecutive transaction-time states (that is, their timestamps meet). This query is nonsequenced because states at two different transaction times are being compared. Consistent with valid time, to access the transaction time in a nonsequenced operation, use TRANSACTIONTIME().

```
NONSEQUENCED TRANSACTIONTIME
SELECT W1.Discoverer, W1.RA_Sec AS Old_Value, W2.RA_Sec AS New_Value,
   END(TRANSACTIONTIME(W1)) AS When_Changed
FROM WDS_TT AS W1, WDS_TT AS W2
WHERE W1.Discoverer = W2.Discoverer
   AND W1.RA_Sec <> W2.RA_Sec
   AND TRANSACTIONTIME(W1) MEETS TRANSACTIONTIME(W2)
```

As valid time and transaction time are orthogonal, they can be easily used together in SQL3.

```
CREATE TABLE WDS_B (
   Discoverer CHAR(7),
   Mag_First DECIMAL(5,2)
) AS VALIDTIME PERIOD(DATE) AND TRANSACTIONTIME
```

With six keywords we get a table with both valid-time and transaction-time support, that is, a bitemporal table.

"What was the history recorded as of January 1, 1994?" is sequenced in valid time ("history") and a timeslice in transaction time ("recorded as of").

```
CREATE VIEW WDS_VT_AS_OF_Jan_1 AS
VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT Discoverer, Mag_First
FROM WDS_B
WHERE TRANSACTIONTIME(WDS_B) OVERLAPS DATE '1994-01-01'
```

"List the corrections made on plates taken in the 1920's" implies sequenced in valid time ("taken in the 1920s") and nonsequenced in transaction time ("corrections", expressed with MEETS).

```
VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT B1.Discoverer, END(TRANSACTIONTIME(B1)) AS When_Changed
FROM WDS_B AS B1, WDS_B AS B2
WHERE B1.Discoverer = B2.Discoverer
   AND B1.Trans_Stop = B2.Trans_Start
   AND TRANSACTIONTIME(B1) MEETS TRANSACTIONTIME(B2)
   AND VALIDTIME(B1) OVERLAPS PERIOD '[1920-01-01 - 1929-12-31]'
```

This query expressed in SQL-92 requires 24 lines of highly complex code.

Current modifications will automatically track the behavior of both valid and transaction time. "A photographic plate indicates that the magnitude of A1248 is 10.5."

```
UPDATE WDS_B
SET Mag_Start = 10.5
WHERE Discoverer = 'A 1248'
```

Current modifications in SQL3 require absolutely no changes when expressed on tables with valid-time, transaction-time, or bitemporal support, as this example illustrates. The same holds for queries, constraints, views, etc. Expressing this modification in SQL-92, manually managing the valid and transaction timestamps in the table, is quite challenging.

Of course, sequenced and nonsequenced (in valid-time) modifications are relevant on a bitemporal table.

To convince yourself of the advantages of the new SQL3 constructs, try expressing the following in SQL-92. All can be formulated in SQL3 in just a few minutes; they would take an SQL-92 expert many *hours* to express in that language.

"LOT_LOC.LOT_ID is a (sequenced) foreign key to LOT," meaning that at every point in time the value of LOT_ID is in LOT at that time.

```
ALTER TABLE LOT_LOC ADD CONSTRAINT VALIDTIME FOREIGN KEY LOT_ID REFERENCES LOT
```

"Give the history of the number of cattle in pen 1."

```
VALIDTIME SELECT COUNT(*)
FROM LOT_LOC
WHERE PEN_ID = 1
```

"Ten head of cattle were added today to lot 219." This is a current update.

```
UPDATE LOT_LOC
SET HD_CNT = HD_CNT + 10
WHERE LOT = 219
```

"When was it recorded that A1248 had a magnitude but no other stars were known of that magnitude?" This is sequenced in transaction time ("when was it recorded").

```
TRANSACTIONTIME SELECT Mag_First
FROM WDS_B AS W1
WHERE Discoverer = 'A 1248'
   AND NOT EXISTS (SELECT * FROM WDS_B AS W2 WHERE W1.Mag_First = W2.Mag_First)
```

"A photographic plate taken in October 1994 indicated that the magnitude of A1248 is 10.5" is sequenced in valid time; the semantics of transaction time are automatically handled in SQL3.

```
VALIDTIME PERIOD '[1994-10-16 - 9999-12-31)'
UPDATE WDS_B
SET Mag_Start = 10.5
WHERE Discoverer = 'A 1248'
```

5.3 Herein the Lesson

To recap, I list five requirements that must be satisfied if a language or DBMS can be claimed to provide temporal support.

1. Both valid time and transaction time are supported, in a compatible and orthogonal manner.

   In particular, the semantics of transaction time, where the state as of a time in the past can be reconstructed, must be guaranteed by the DBMS.

2. Upward compatibility is ensured.

   Existing constructs applied to nontemporal data should operate exactly as before. This requirement is fairly easy to satisfy.

3. Temporal upward compatibility is ensured.

   This means that an existing nontemporal application will not be broken when temporal support is added to a table, say via an ALTER TABLE statement. No changes to application code should be required when the history of the enterprise (valid time) or the sequence of changes to the data (transaction time), or both, is retained. This implies, for example, that a conventional query on tables with temporal support should be interpreted as a current query.

4. Sequenced variants should be easy to express for *all* constructs of the language, including queries, modifications, views, assertions and constraints, and cursors.

   In particular, complex rewritings of the statement should not be necessary.

5. Nonsequenced variants should also be easy to express.

   In part, such variants enable data with temporal support to be converted to and from data without temporal support.

The SQL3 constructs discussed here satisfy these requirements.

1. Valid time can be added to a table via AS VALIDTIME PERIOD; transaction time can be added with AS TRANSACTIONTIME. Only current modifications are allowed in transaction time, to ensure that timeslices will be correct. Either kind of time can be used individually, or together, forming a bitemporal table.

2. SQL/Temporal is defined as an upward compatible extension of the other parts of SQL3.

3. All conventional queries (modifications, views, assertions, constraints, cursors) on tables with temporal support are interpreted as current queries (resp., modifications, etc.) As an example, when valid-time support was added to the NICUStatus table, the existing code of this application, perhaps tens of thousands of lines, did not require a single change.

4. A query can be converted to a sequenced query in SQL3 simply by prepending the keyword VALIDTIME. This also holds for modifications (e.g., VALIDTIME UPDATE), views (e.g., CREATE VIEW AS VALIDTIME SELECT), and constraints (e.g., VALIDTIME UNIQUE). And of course this also applies to transaction time, via the TRANSACTIONTIME keyword.

5. Nonsequenced statements require the additional keyword NONSEQUENCED. The valid timestamp associated with a row is accessible via the function VALIDTIME() and the transaction timestamp via TRANSACTIONTIME().

As we have shown with sample SQL3 statements, these proposed constructs (three new reserved words, VALIDTIME, TRANSACTIONTIME and NONSEQUENCED, in addition to those already in SQL/Temporal) can greatly simplify application development, often reducing the amount of SQL code that needs to be written by a factor of 10 or more, while improving the comprehensibility of that code. We look forward to the day when these constructs are in the SQL standard, and even more importantly, when they are supported by products.

5.4 Building the Standard

SQL-86 and SQL-89 have no notion of time. SQL-92 added datetime and interval data types, though no product has yet been validated for conformance to this standard (some products *have* been validated at the entry level, which does not include the temporal data types). However, it has long been recognized in the temporal database research community, and as the case studies in this special series have illustrated, that these data types alone are inadequate. Momentum for a temporal extension to SQL designed by that community first became evident at the Workshop on an Infrastructure for Temporal Databases, held in Arlington, Texas in June, 1993.

The TSQL2 committee was subsequently formed, producing a preliminary language specification the following January. The final version was completed in September, 1994, and a book describing the language and examining in detail its underlying design decisions was released at the VLDB International Workshop on Temporal Databases in Zurich in September, 1995 [3].

The ANSI and ISO SQL3 committees became involved in late 1994. A new part to SQL3, termed SQL/Temporal, was proposed and formally approved by the SQL3 International Organization for Standardization (ISO) in Ottawa in July, 1995 as Part 7 of the SQL3 draft standard. Jim Melton agreed to edit this new part. The first task was to define a PERIOD data type, which is now included in Part 7.

Discussions then commenced on adding further temporal support. Two change proposals resulted, one on valid time support and one on transaction time support [4, 5]. These change proposals have been unanimously approved by the ANSI SQL3 committee (ANSI X3H2) for consideration by the ISO SQL3 committee (ISO/IEC JTC 1/SC 21/WG 3 DBL). The full story may be found at <http://www.cs.arizona.edu/people/rts/tsql2.html>.

In the meantime, the SQL committees decided to focus on parts 1, 2, 4, and 5 of the SQL3 draft standard. These parts are expected to be finalized as an international standard next year. At that time, the committees will revisit the other parts and move them through the exhaustive process towards standardization.

6 Code Samples

Code samples for all the case studies, in a variety of DBMSs, can be found at <www.arizona.edu/people/rts/DBPD>.

7 Acknowledgments

Database Programming and Design can be found on the web at <http://www.dbpd.com>. This publication has merged with *DBMS Magazine* to form *Intelligent Enterprise*, at <http://www.intelligententerprise.com/>. The articles included in the present document are reprinted (with slight changes) with permission and are copyright ©1998 Miller Freeman, Inc.

I thank Dr. Brad De Groot for background information on feed lot databases, Jim Melton for pointing out the PSM solution for a valid-time join, and James Scotti of the University of Arizona's Lunar and Planetary Laboratory for help interpreting star catalogues.

8 About the Author

Richard Snodgrass is a professor of computer science at the University of Arizona. He chairs ACM SIGMOD, has written many papers and several books on temporal databases, and consults on designing and implementing time-varying databases. He is working with the ANSI and ISO SQL3 committees to add temporal support to that language. His book, **Developing Time-Oriented Applications in SQL**, will be published by Morgan Kaufmann Publishers early next year.

References

[1] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes and S. Jajodia [eds], "A Glossary of Temporal Database Concepts," *ACM SIGMOD Record*, 23(1):52–64, March, 1994.

[2] G. Ozsoyoglu and R. T. Snodgrass, "Temporal and Real-Time Databases: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August, 1995.

[3] R. T. Snodgrass (editor), I. Ahn, G. Ariav, D. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Kaefer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo and S.M. Sripada. **The Temporal Query Language TSQL2**. Kluwer Academic Publishers, 1995.

[4] R. T. Snodgrass, M. H. Boehlen, C. S. Jensen and A. Steiner. Adding Valid Time to SQL/Temporal. Change proposal, ANSI X3H2-96-501r2, ISO/IEC JTC 1/SC 21/WG 3 DBL-MAD-146r2, November 1996. At URL: <ftp://ftp.cs.arizona.edu/tsql/tsql2/sql3/mad146.pdf>

[5] R. T. Snodgrass, M. H. Boehlen, C. S. Jensen and A. Steiner. Adding Transaction Time to SQL/Temporal. Change proposal, ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/WG3 DBL MAD-147r2, November 1996. At URL: <ftp://ftp.cs.arizona.edu/tsql/tsql2/sql3/mad147.pdf>

[6] A. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass [eds], **Temporal Databases: Theory, Design, and Implementation**, Database Systems and Applications Series, Benjamin/Cummings Pub. Co., Redwood City, CA, March 1993, 633+xx pages.

[7] V. J. Tsotras and A. Kumar, "Temporal Database Bibliography Update," *ACM SIGMOD Record*, 25(1):41–51, March, 1996.

[8] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari, **Advanced Database Systems**, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997, 574+xvii pages.