Chapter 5

Overview of Temporal Databases

Time is an important aspect of all real-world phenomena. Database applications must capture the time-varying nature of the phenomena they model.

As one example, every university maintains a temporal database recording the courses taken by each student and the grades that student received. This information certainly varies over time: courses are taken during a particular semester, and indeed a course may be taken several times by a student. While the registrar may be most interested in the current semester—what students are signed up for what classes—graduating students are probably more interested in getting printouts of their entire transcript. And department heads often look at enrollment statistics over time to better plan future staffing.

As another example, advertising agencies plan very carefully when advertisements will run. Television advertisements are especially expensive (an advertisement during the Super Bowl costs about $2 million a minute; advertising during a prime-time show can cost $100,000 to $600,000 a minute); print media are also costly. Agencies store media plans in databases, recording which advertisements are to appear when, the costs involved, and an estimate of the number of people, in various categories, that will see each advertisement. Analyses are run against this information, such as determining the advertising budget over time (is it relatively flat, or are there peaks and valleys?), estimating the total number of people in each category that will be exposed to the product, and evaluating the integration with print advertisements (do television advertisements run prior to, during, or after...
5. OVERVIEW OF TEMPORAL DATABASES

the appearance of print advertisements?). Both the underlying information and the queries over this information are inherently temporal.

Conventional (nontemporal) databases represent the state of an enterprise at a single moment in time. Although the contents of the database continue to change as new information is added, these changes are viewed as modifications to the state, with the old, out-of-date data being deleted from the database. The current content of the database may be viewed as a snapshot of the enterprise. As we will see shortly in a case study, when a conventional database is used, the attributes involving time are manipulated solely by the application programs, with little help from the database management system (DBMS).

From the academic database example, the current state contains information about the semester in progress. Queries and updates concern this semester. Information on previous semesters can be stored by associating a column identifying the relevant semester. However, in a nontemporal DBMS, this column must be explicitly manipulated by the user. Obtaining enrollment statistics over time requires complex application code.

A temporal database is one that supports some aspect of time. We will shortly encounter more specific characterizations that concern the kind(s) of time supported. Queries over previous states are easy to specify. Also, modifications to previous states (if an error is detected, or if more information becomes available) and to future states (for planning purposes) are also easier to express using a temporal DBMS.

A temporal DBMS allows sophisticated queries over time to be stated. As an example, using the media planning database, we may want to determine the well-established shows, to potentially use for our advertising. More specifically, we request all shows broadcast by NBC that ran continuously for at least two years, as well as the day that they began that run. A more sophisticated query would be to calculate the advertising budget for each month for the major television networks (ABC, CBS, NBC, Fox). Such queries using a temporal query language are but a few lines long.

Almost all database applications concern time-varying information. In fact, it is difficult to identify applications that do not involve the management of time-varying data. The advantages provided by built-in temporal support include higher-fidelity data modeling, more efficient application development, and a potential increase in performance.

5.1 A CASE STUDY

5.1 A Case Study

We first demonstrate the need for temporal databases with a case study that illustrates the pitfalls of using a conventional DBMS to underlie a time-varying application.

The University of Arizona's Office of Appointed Personnel (OAP) has information concerning university employees in a database; this information includes the employee's name, their current salary, and their current title. In the relational model, this can be represented by a simple, three-column relation:

Employee(Name, Salary, Title)

Each tuple (row) of this relation provides information on one employee; different tuples are associated with distinct employees.

Given this relation, finding the employee's salary is easy when a relational query language such as SQL is used:

Example 5.1 What is Bob's salary?

SELECT Salary
FROM Employee
WHERE Name = 'Bob'

Now the OAP wishes to record the date of birth. To do so, a column is added to the relation, yielding the following schema:

Employee(Name, Salary, Title, DateOfBirth DATE)

Finding the employee's date of birth is analogous to determining the salary:

Example 5.2 What is Bob's date of birth?

SELECT DateOfBirth
FROM Employee
WHERE Name = 'Bob'

This illustrates the (limited) temporal support available in SQL (more precisely, in the SQL-92 standard, as well as in all major commercial DBMSs), that of the column type DATE. As we will see later, other temporal types are available, but they are not sufficient for ease in querying time-varying data. Rather, a temporal query language is required.

The OAP wishes to record the employment history. To do so, they append two columns, one indicating when the information in the tuple became valid, the other indicating when the information was no longer valid.
Employee (Name, Salary, Title, DateofBirth, Start DATE, Stop DATE)

To the data model, these new columns are identical to DateofBirth. However, their presence has far-ranging consequences.

5.1. Temporal Projection

To find the employee’s current salary, things are more difficult:

Example 5.3 What is Bob’s current salary?

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Title</th>
<th>DateofBirth</th>
<th>Start</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>60000</td>
<td>Assistant Provost</td>
<td>1945-04-09</td>
<td>1995-01-01</td>
<td>1995-06-01</td>
</tr>
<tr>
<td>Bob</td>
<td>70000</td>
<td>Assistant Provost</td>
<td>1945-04-09</td>
<td>1995-06-01</td>
<td>1995-10-01</td>
</tr>
<tr>
<td>Bob</td>
<td>70000</td>
<td>Provost</td>
<td>1945-04-09</td>
<td>1995-10-01</td>
<td>1996-02-01</td>
</tr>
<tr>
<td>Bob</td>
<td>70000</td>
<td>Professor</td>
<td>1945-04-09</td>
<td>1996-02-01</td>
<td>1997-01-01</td>
</tr>
</tbody>
</table>

This query is more complicated than the previous one. The culprit is obviously the two new columns.

The OAP wants to distribute to each employee their salary history. Specifically, for each person, the maximal periods of time for each salary needs to be determined. Unfortunately, this is very difficult in SQL. An employee could have arbitrarily many title changes between salary changes, as shown below:

Example 5.4 What is Bob’s salary history (first attempt)?

```sql
CREATE TABLE Temp(Salary, Start, Stop)
AS SELECT Salary, Start, Stop
FROM Employee
WHERE Name = 'Bob';
```

```
repeat
UPDATE Temp T1
SET (T1.Start) = (SELECT MAX(T2.Start)
FROM Temp AS T2
WHERE EXISTS (SELECT *
FROM Temp AS T2
```

```
until no tuples updated;
```

```
DELETE FROM Temp T1
WHERE EXISTS (SELECT *
FROM Temp AS T2
WHERE T1.Salary = T2.Salary
OR (T1.Start >= T2.Start AND T1.Stop < T2.Stop))
```

Assume that Bob had many tuples with the same salary, but with different titles. The Temp relation would initially contain nonoverlapping time periods:

```
| Time Periods |

After the first iteration of the repeat-until loop, the Temp relation would contain:

```
| Time Periods |

Here is one possible solution. The intuition is that we find those time periods with the same salary value that overlap or are adjacent, and merge those periods by extending the earlier period. This process is repeated until maximal periods are constructed. At that point, the nonmaximal periods are removed. This process is termed coalescing.
Note how the Stop time is extended when a value-equivalent tuple (one with identical values for the nontimestamp attributes, in this case the Salary attribute) meets or overlaps it.

After the second iteration, some periods are further extended:

```
.-                -
```

The next iteration does not change any stop time, and so the repeat-until loop is terminated. The DELETE statement removes the nonmaximal value-equivalent periods, retaining only the last one shown above.

One problem with this approach is that it uses a non-SQL statement, the repeat-until loop. For a time, it was thought impossible to express this query completely in SQL. A solution was discovered just a few years ago, involving complex, multiply nested NOT EXISTS subqueries:

Example 5.5  What is Bob's salary history (entirely in SQL)?

```sql
CREATE TABLE Temp(Salary, Start, Stop)
AS SELECT Salary, Start, Stop
FROM Employee
WHERE Name = 'Bob';

SELECT DISTINCT F.Salary, F.Start, L.Stop
FROM Temp AS F, Temp AS L
WHERE F.Start < L.Stop
AND F.Salary = L.Salary
AND NOT EXISTS (SELECT *
FROM Temp AS M
WHERE M.Salary = F.Salary
AND F.Start < M.Start AND M.Start < L.Stop
AND NOT EXISTS (SELECT *
FROM Temp AS T1
WHERE T1.Salary = F.Salary
AND T1.Start < M.Start AND M.Start <= T1.Stop))
AND NOT EXISTS (SELECT *
FROM Temp AS T2
WHERE T2.Salary = F.Salary
AND ((T2.Start < F.Start AND F.Start <= T2.Stop) OR
(T2.Start < L.Start AND L.Start < T2.Stop)))
```

In this query, we search for two (possibly the same) value-equivalent tuples (represented by the correlation names F, for first, and L, for last) defining start point F.Start and end point L.Stop of a coalesced tuple. The first NOT EXISTS ensures that there are no holes between F.Start and L.Stop (i.e., no time points where the respective fact does not hold). This guarantees that all start points M.Start between F.Start and L.Stop of value-equivalent tuples are extended (toward F.Start) by another value-equivalent tuple. This is illustrated below:

```
 F  T1  L
--------
```

In this subclause, T1 may in fact be F. It may also be the case that F itself overlaps L, in which case the NOT EXISTS is certainly true.

The second NOT EXISTS ensures that only maximal periods result (i.e., F and L cannot be part of a larger value-equivalent tuple T2).

A third alternative is to use SQL only to open a cursor on the relation. A linked list of periods is maintained, each with a salary. This linked list should be initialized to empty.

Example 5.6  What is Bob's salary history (using a cursor)?

```sql
DECLARE emp.cursor CURSOR FOR
SELECT Salary, Title, Start, Stop
FROM Employee
WHERE Name = 'Bob';

OPEN emp.cursor;

loop:
  FETCH emp.cursor INTO :salary, :start, :stop;
  if no data returned then goto finished;
  find position in linked list to insert this information;
  goto loop;

finished:
CLOSE emp.cursor;
iterate through linked list, printing out dates and salaries
```

The linked list is unnecessary if the cursor is ORDER BY Start.
In all cases, the query is quite complex for such a simple English equivalent. The reason is that SQL is a nontemporal query language. The language has no facilities for timestamped tuples.

The query is trivial in TSQL2, a temporal query language that will be discussed in depth later:

Example 5.7 What is Bob's salary history (in TSQL2)?

```sql
SELECT Salary 
FROM Employee 
WHERE Name = 'Bob'
```

5.1.2 Temporal Join

A more drastic approach avoids the problem in SQL of extracting the salary history by reorganizing the schema to separate salary, title, and date of birth information.

Employee1 (Name, Salary, Start DATE, Stop DATE)
Employee2 (Name, Title, Start DATE, Stop DATE)

With this change, getting the salary information is now easy.

Example 5.8 What is Bob's salary history (using Employee1)?

```sql
SELECT Salary, Start, Stop 
FROM Employee1 
WHERE Name = 'Bob'
```

But what if the OAP wants a relation of salary/title periods? Using SQL, the query must do a case analysis of how each tuple of Employee1 overlaps each tuple of Employee2; there are four possible cases.

Example 5.9 Provide the salary and department history for all employees

```sql
FROM Employee1, Employee2 
WHERE Employee1.Name = Employee2.Name 
AND Employee1.Start <= Employee1.Start 
AND Employee1.Stop <= Employee2.Stop
UNION 
FROM Employee1, Employee2 
WHERE Employee1.Name = Employee2.Name 
AND Employee1.Start > Employee2.Start 
AND Employee1.Stop < Employee1.Stop 
AND Employee1.Start < Employee2.Start
```

In the first case, the period associated with the Employee2 tuple entirely contains the period associated with the Employee1 tuple. Since we are interested in those times when both the salary and the department are valid, the intersection of the two periods is the contained period, that is, the period from Employee1.Start to Employee1.Stop:

```
  Employee1
    +--------+-----+-----+-----+-----+
    | Salary | Start| Stop| Dept| Name|
    +--------+-----+-----+-----+-----+
```

In the second case, neither period contains the other:

```
  Employee1
    +--------+-----+-----+-----+-----+
    | Salary | Start| Stop| Dept| Name|
    +--------+-----+-----+-----+-----+
```

The other cases similarly identify the overlap of the two periods.

While this query is not as complex as those given before, it still requires care to get the eleven inequalities and the four select lists correct.
5. OVERVIEW OF TEMPORAL DATABASES

In a temporal query language such as TSQL2, performing a temporal join is just what one would expect:

Example 5.10 Provide the salary and department history for all employees (in TSQL2)

```
SELECT Employee1.Name, Salary, Dept
FROM Employee1, Employee2
WHERE Employee1.Name = Employee2.Name
```

5.1.3 Summary

Time-varying data is common, and applications that manage such data abound. However, nontemporal DBMSs and their query languages provide inadequate support for such applications. If a temporal DBMS is used, the data model more accurately reflects reality, SQL queries are much simpler, and significantly less application code is required. This enables developers to more easily write, debug, and maintain applications.

In the remainder of this part of the book, we will discuss the foundations of temporal databases, survey the many temporal data models that have been developed, delve into the TSQL2 temporal query language, and discuss implementation strategies for temporal DBMSs.

5.2 The Time Domain

Here we focus on time itself: how it is modeled and how it is represented. Section 5.4 will then combine time with facts, to model time-varying information.

Models of time in a temporal logic represent time as an arbitrary set of instants with an imposed partial order. Additional axioms introduce other, more refined models of time. For example, linear time can be specified by adding an axiom imposing a total order on this set. In the linear model, time advances from the past to the future in a step-by-step fashion. In the branching model, also termed the possible futures or hypothetical model, time is linear from the past to now, where it then divides into several time lines, each representing a potential sequence of events. Along any future path, additional branches may exist. The structure of branching time is a tree rooted at now. Generalizations allow branches in the past, or allow branches to join. Recurrent processes may be associated with a cyclic model of time. An example is a week, in which each day recurs every seven days.

Axioms may also be added to temporal logics to characterize the density of the time line. Combined with the linear model, discrete models of time are isomorphic to the natural numbers, implying that each point in time has a single successor. Dense models of time are isomorphic to either the rationals or the reals: between any two moments of time another moment exists. Continuous models of time are isomorphic to the reals, that is, they are both dense and, unlike the rationals, contain no "gaps."

In the continuous model, each real number corresponds to a "point" in time; in the discrete model, each natural number corresponds to a nondecomposable unit of time with an arbitrary duration. Such a nondecomposable unit of time is referred to as a chronon. A chronon is the smallest duration of time that can be represented in this model. It is not a point, but a line segment on the time line.

Although time itself is perceived by most to be continuous, the discrete time model is generally used. Several practical arguments justify this choice. First, measures of time are inherently imprecise. Clocking instruments invariably report the occurrence of events in terms of chronons, not time "points." Hence, events, even so-called instantaneous events, can at best be measured as having occurred during a chronon. Second, most natural language references to time are compatible with the discrete time model. For example, when we say that an event occurred at 4:30 PM, we usually don't mean that the event occurred at the "point" in time associated with 4:30 PM, but at some time in the chronon (perhaps minute) associated with 4:30 PM. Third, the concepts of chronon and period allow us to naturally model events that are not instantaneous but have duration. Finally, any implementation of a data model with a temporal dimension will of necessity have to have some discrete encoding for time.

Axioms can also be placed on the boundedness of time. Time can be bounded orthogonally in the past and in the future. A finite encoding implies bounds from the left (i.e., the existence of a time origin) and from the right. Models of time may include the concept of distance, though most temporal logics do not do so.

Finally, one can differentiate relative time from absolute time (more precise terms are unanchored and anchored). For example, "9 AM, January 1, 1996" is an absolute time, and "9 hours" is a relative time. This distinction, though, is not as crisp as we would hope, because absolute time is with respect to another time (in this example, midnight, January 1, AD 1), termed an anchor. Relative time can be distinguished from distance in that the former has a direction. For example, you could envision a relative time of -9 hours, but distance is unsigned.
5.3 Time Data Types

Several temporal data types have proven useful. The most basic is a time 
instant, which is a particular chronon on the time line. An event is an 
instantaneous fact, that is, something occurring at an instant. The event 
ocurrence time of an event is the instant at which the event occurs in the 
real world.

SQL-92 provides three instant data types: DATE (a particular day, with 
a year in the range AD 1–9999), TIME (a particular second within a range of 
24 hours), and TIMESTAMP (a particular fraction of a second, defaulting 
to microsecond, of a particular day).

A time period is the time between two instants. In some of the litera-
ture, this notion is called a time interval, but this usage conflicts with 
the SQL-92 data type INTERVAL, which is a different concept altogether.
SQL-92 does not include periods, but periods are now part of the evolving 
SQL3 specification.

A time interval is a directed duration of time, that is, an amount of 
time with a known length, but not specific starting or ending instants. A 
positive interval denotes forward motion of time, toward the future. SQL-92 
supports two kinds of intervals, month-year and second-day intervals.

Two final temporal data types are instant sets, which are (logically!) sets 
of instants, and temporal elements, which are finite unions of periods.

Temporal types must be representable. A bounded discrete representa-
tion, as an integer count of the instants since the origin, is the simplest 
option. A bounded dense representation is also not difficult to manage, as 
all rationals may be expressed as the ratio between two integers. A floating 
point representation may also be employed. A continuous representation is 
the most difficult to implement.

5.4 Associating Facts with Time

The previous sections discussed the time domain itself. We now turn to 
associating time with facts.

5.4.1 Dimensionality

In the context of databases, two time dimensions are of general interest: the 
valid time dimension and the transaction time dimension.

Valid time concerns the time a fact was true in reality. The valid time of 
an event is the time at which the event occurred in the real world, indepen-
dent of the recording of that event in some database. Valid times can also 
be in the future, if it is expected that some fact will be true at a specified 
time after now.

Transaction time concerns the time the fact was present in the database 
as stored data. The transaction time (a period) of a fact identifies the 
transaction that inserted the fact into the database and the transaction that 
removed this fact from the database.

These two dimensions are orthogonal. A data model supporting neither 
is termed snapshot, as it captures only a single snapshot in time of both 
the database and the enterprise that the database models. A data model 
supporting only valid time is termed valid-time, one that supports only transaction time is termed transaction-time, and one that supports both valid and transaction time is termed bitemporal. Temporal is a generic term implying 
some kind of time support.

Figure 5.1 illustrates the structure of a (three-column) conventional re-
lation. A relation consists of a number of tuples (rows), each having the 
same number of attribute values (columns). Each tuple captures a fact that 
is currently thought to be true in the modeled reality. As reality changes, 
the relation changes, with tuples added, removed, or modified.

Figure 5.2 illustrates the structure of a transaction-time relation, which 
is a sequence of snapshot states, indexed over transaction time. The relation 
started out as an empty relation. A transaction inserted three tuples, re-
sulting in the first state being appended to the sequence, with an associated 
transaction time of the commit time of that transaction. A later transaction 
inserted one tuple, resulting in the second state being appended. A subse-
quent transaction then deleted the first tuple and inserted yet another tuple, 
resulting in the third state.

This figure emphasizes the semantics of transaction time. Unlike snap-
shot relations, transactions do not alter existing data in transaction-time 
relations. Rather, the change is made to the current snapshot state, re-
sulting in a new snapshot state that is appended to the relation. In this 
sense, transaction-time relations are append-only, and thus amenable to be-
one added in the first transaction, was changed to a somewhat later time (presumably the original starting time was incorrect) and a tuple (the top one) was inserted. Each update operation involves copying the valid-time state, then applying the update to the newly created state. Of course, less redundant representations than the one shown are possible.

As with transaction-time relations, bitemporal relations are append-only and support transaction-time queries. As with valid-time relations, bitemporal relations support valid-time queries and permit facts to be true in the past and in the future.

While valid time may extend into the future, transaction time is defined only until now. Specifically, transaction time starts when the database is created (before which time, nothing was stored) and doesn’t extend past now (no facts are known to have been stored in the future). Changes to the database state are required to be stamped with the current transaction time. As the database state evolves, transaction times grow monotonically. In contrast, successive transactions may mention widely varying valid times. For instance, the fourth transaction in Figure 5.4 added information to the database that was transaction timestamped with time 4, while changing a valid time of one of the tuples to 2.

The two time dimensions are not homogeneous; transaction time has a different semantics than valid time. Valid and transaction time are orthogonal, though there are generally some application-dependent correlations between the two times. As a simple example, consider the situation where a fact is recorded as soon as it becomes valid in reality. In such a specialized bitemporal database, termed degenerate, the valid and transaction times of a fact are identical. As another example, if a cloud cover measurement is recorded at most 2 days after it was valid in reality, and if it takes at least 6 hours from the measurement time to record the measurement, then such a relation is characterized as “delayed strongly retroactively bounded with bounds 6 hours and 2 days.”
### 5. OVERVIEW OF TEMPORAL DATABASES

<table>
<thead>
<tr>
<th>Data model name</th>
<th>Temporal dimension(s)</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting Data Model</td>
<td>Both</td>
<td>ADM</td>
</tr>
<tr>
<td>Temporally Oriented Data Model</td>
<td>Both, Valid, Both</td>
<td>Arles, Bouslama, Bhargava</td>
</tr>
<tr>
<td>Bitemporal Conceptual Data Model</td>
<td>Both</td>
<td>BCDM</td>
</tr>
<tr>
<td>Time Relational Model</td>
<td>Both</td>
<td>Ben-Zvi</td>
</tr>
<tr>
<td>DATA</td>
<td>Transaction</td>
<td>DM/T</td>
</tr>
<tr>
<td>Extensional Data Model</td>
<td>Both</td>
<td>EDM</td>
</tr>
<tr>
<td>Homogeneous Relational Model</td>
<td>Valid</td>
<td>Gadia-1, Gadis-2</td>
</tr>
<tr>
<td>Heterogeneous Relational Model</td>
<td>Valid</td>
<td>HDM</td>
</tr>
<tr>
<td>Historical Data Model</td>
<td>Valid</td>
<td>HRDM</td>
</tr>
<tr>
<td>Historical Relational Data Model</td>
<td>Valid, Transaction</td>
<td>Lomet, Jones</td>
</tr>
<tr>
<td>Temporal Relational Model</td>
<td>Valid</td>
<td>Lornettoz</td>
</tr>
<tr>
<td></td>
<td>Valid</td>
<td>Lums</td>
</tr>
<tr>
<td>Temporal Relational Model</td>
<td>Both</td>
<td>McKenzie</td>
</tr>
<tr>
<td>Temporal Data Model</td>
<td>Valid, Valid, Valid</td>
<td>Navathe, Sadeghi, Sargs</td>
</tr>
<tr>
<td>Time Oriented Database Model</td>
<td>Valid</td>
<td>Seger, Snodgrass, Tausel</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Yau</td>
</tr>
</tbody>
</table>

*Table 5.1: Temporal relational data models*

#### 5.4. ASSOCIATING FACTS WITH TIME

<table>
<thead>
<tr>
<th>Data model name</th>
<th>Temporal dimension(s)</th>
<th>Identifier</th>
<th>Transaction representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRIS</td>
<td>Both</td>
<td>Caruso</td>
<td>Chronon, identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Version hierarchy</td>
</tr>
<tr>
<td>MATISSE</td>
<td>Transaction</td>
<td>MATISSE</td>
<td>Chronon, identifier</td>
</tr>
<tr>
<td>OODAPLEX</td>
<td>Arbitrary</td>
<td>OODAPLEX</td>
<td>Arbitrary, N/A (identification)</td>
</tr>
<tr>
<td>OSAM*</td>
<td>Valid</td>
<td>OSAM*</td>
<td>N/A (identification)</td>
</tr>
<tr>
<td>VVM</td>
<td>Transaction</td>
<td>VVM</td>
<td>N/A (identification)</td>
</tr>
<tr>
<td>Postgres</td>
<td>Transaction</td>
<td>Postgres</td>
<td>N/A (identification)</td>
</tr>
<tr>
<td></td>
<td>Both</td>
<td>Sciore-1</td>
<td>N/A (identification)</td>
</tr>
<tr>
<td>TEDM</td>
<td>Valid</td>
<td>TEDM</td>
<td>N/A (identification)</td>
</tr>
<tr>
<td>TIGUKAT</td>
<td>Both</td>
<td>TIGUKAT</td>
<td>N/A (identification)</td>
</tr>
<tr>
<td>TMAD</td>
<td>Valid</td>
<td>TMAD</td>
<td>N/A (identification)</td>
</tr>
<tr>
<td>Temporal</td>
<td>Both</td>
<td>TOODM</td>
<td>Temporal element</td>
</tr>
<tr>
<td>Object-Oriented</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Model</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 5.2: Temporal object-oriented data models*

We classify the extant object-oriented temporal data models in Table 5.2. The last column will be discussed shortly. Those with "arbitrary" indicated in the temporal dimensions and transaction timestamp representation columns support time with user- or system-provided classes; hence, anything is possible. N/A denotes “not applicable.”

### 5.4.2 Underlying Data Model

Time has been added to many data models: the entity-relationship model, semantic data models, knowledge-based data models, and deductive data models. However, by far the majority of work in temporal databases is based on the relational and object-oriented models. For this reason, we focus on these two data models in our subsequent discussion.

Table 5.1 lists most of the temporal relational data models that have appeared in the literature. Some models are defined only over valid time or transaction time; others are defined over both. The last column indicates a short identifier that denotes the model; the table is ordered on this column.1

1If the model has not been given a name, we use the name of the first designer of the model as an identifier; this is also done for models with identical acronyms. Citations of papers describing these models may be found in the bibliographic notes at the end of this chapter.

Valid Time

These models may be compared along the valid-time dimension by asking two basic questions: how is valid time represented and how are facts associated with valid time. Table 5.3 categorizes most of the data models along these two aspects. We do not include the OODAPLEX, Sciore-1, and TIGUKAT data models, as these two aspects are arbitrarily specifiable in these models.

Valid times can be represented with single chronon identifiers (i.e., instantaneous timestamps), with periods (i.e., as period timestamps), or as valid-time elements, which are finite sets of periods. Valid time can be associated with
individual attribute values, with groups of attributes, or with an entire tuple or object. Finally, constraints over the integers (or reals) may be used to express the times at which a tuple is valid. The solutions of the constraints are time points. The relational algebra can then be extended by manipulating these constraints. Other alternatives, such as associating valid time with sets of tuples (i.e., relations) or with object graphs (i.e., a set of objects, with an attribute of one object referencing another object in the set, forming a connected graph), have not been incorporated into any of the proposed data models, primarily because they introduce considerable data redundancy.

Transaction Time

The same general issues are involved in transaction time, but there are about three times as many alternatives. The choices made in the various data models are characterized in Table 5.4. OODAPLEX is not included, as it can support virtually any of these options (while that is also possible in TIGUKAT, specific support for versioning has been added to the data model and language). Transaction time may be represented with the following alternatives:

- The transaction timestamp may be a single chronon, which implies that tuples inserted on each transaction signify the termination (logical deletion) of previously current tuples with identical keys, with the timestamps of these previously recorded tuples not requiring change.

  - The timestamp may be a period. A newly inserted tuple would be associated with the period starting at now and ending at the special value U.C. (until changed).

  - The timestamp may consist of three chronons. Ben-Zvi's model records (1) the transaction time when the valid start time was recorded, (2) the transaction time when the valid stop time was recorded, and (3) the transaction time when the tuple was logically deleted.

  - The timestamp may be a transaction-time element, which is a set of periods.

More detail on the representation of “Other” may be found in the last column of Table 5.2. Specifically, those data models supporting versions often allow arbitrary user-supplied identifiers to be associated with versions. One model even allows an entire version hierarchy to be associated with a version.
5.4.3 Representative Data Models

To ground this discussion, let’s examine five representative models. One of the simplest is Segov’s valid-time data model, in which tuples are time-stamped with the instant that the tuple became valid. This allows the history of the attribute values of a key to be succinctly captured. In the following relation instance, we see that Eric started working in the shoe department on June 1 (in these examples, we omit the month and year from the timestamp). He moved to the book department on June 6, and returned to the shoe department on June 11. He resigned on June 13; this requires a separate tuple, with null values for all the non-key attributes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Dept</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eric</td>
<td>Shoe</td>
<td>1</td>
</tr>
<tr>
<td>Eric</td>
<td>Book</td>
<td>6</td>
</tr>
<tr>
<td>Eric</td>
<td>Shoe</td>
<td>11</td>
</tr>
<tr>
<td>Eric</td>
<td>Null</td>
<td>13</td>
</tr>
</tbody>
</table>

This data model can use such a simple timestamp because it does not permit multiple values at any point in time. By using period timestamps, as for example in Sarda’s data model, multiple values can be accommodated. The following shows the same information as above, in a period-timestamped model.

<table>
<thead>
<tr>
<th>Name</th>
<th>Dept</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eric</td>
<td>Shoe</td>
<td>[1-5]</td>
</tr>
<tr>
<td>Eric</td>
<td>Book</td>
<td>[6-10]</td>
</tr>
<tr>
<td>Eric</td>
<td>Shoe</td>
<td>[11-13]</td>
</tr>
</tbody>
</table>

Note that null values are not required in Sarda’s model when an employee resigns.

Several of the models timestamp attribute values instead of tuples. This allows more history to be captured in a single tuple. In the HRDM, attribute values are functions from time to a value domain:

<table>
<thead>
<tr>
<th>Name</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 → Eric</td>
<td>1 → Shoe</td>
</tr>
<tr>
<td>12 → Eric</td>
<td>5 → Shoe</td>
</tr>
<tr>
<td></td>
<td>6 → Book</td>
</tr>
<tr>
<td></td>
<td>10 → Book</td>
</tr>
<tr>
<td></td>
<td>11 → Shoe</td>
</tr>
<tr>
<td></td>
<td>12 → Shoe</td>
</tr>
</tbody>
</table>

Figure 5.5: The extensional data model

Eric’s entire employment history is captured in a single tuple. Another advantage of attribute value timestamping is that attributes that vary independently, termed asynchronous attributes, do not require an additional tuple when an attribute value changes.

The above data models are all valid-time models. As a simple example of a bitemporal data model, the extensional data model timestamps each tuple with a single valid-time chronon and a single transaction-time chronon.
5. OVERVIEW OF TEMPORAL DATABASES

5.5 TEMPORAL QUERY LANGUAGES

A data model consists of a set of objects with a specified structure, a set of constraints on those objects, and a set of operations on those objects. In the two previous sections we have investigated in detail the structure of, and constraints on, the objects of temporal databases. Here, we complete the picture by discussing the operations, specifically temporal query languages.

Many temporal query languages have been proposed. In fact, it seems that each researcher feels it necessary to define a new data model and query language.

Table 5.5 lists the major temporal query language proposals to date. The underlying data model is a reference to Table 5.1. The next column lists the conventional query language the temporal proposal is based on. Most of the query languages have a formal definition.

Table 5.6 lists the object-oriented query languages that support time. Note that many "nested" relational query languages and data models, such as HQuel, HTRDM, HTQul, TempSQL, and TBE, have features that might be considered to be object-oriented.

The data model and conventional query language on which the temporal query language is based are identified in the second and third columns. The fourth column indicates whether the language has been implemented. It is rare for an object-oriented query language to have a formal semantics. Also in contrast to temporal relational query languages, most object-oriented query languages have been implemented.

5.6 Summary

A temporal data model attempts to simultaneously satisfy many goals. It should capture the semantics of the application to be modeled in a clear and concise fashion. It should be a consistent, minimal extension of an existing data model, such as the relational model. It is best if the tempo-
5. OVERVIEW OF TEMPORAL DATABASES

<table>
<thead>
<tr>
<th>Name</th>
<th>Underlying data model</th>
<th>Based on</th>
<th>Formal semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>HQL</td>
<td>Sadeghl</td>
<td>DEAL</td>
<td>✓</td>
</tr>
<tr>
<td>HQrel</td>
<td>Tansel</td>
<td>Quel</td>
<td>✓</td>
</tr>
<tr>
<td>HSQL</td>
<td>Sarda</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>HTQuel</td>
<td>Gadita-1</td>
<td>Quel</td>
<td>✓</td>
</tr>
<tr>
<td>IXSQL</td>
<td>Lorentzos</td>
<td>SQL-92</td>
<td>✓</td>
</tr>
<tr>
<td>Legol 2.0</td>
<td>Jones</td>
<td>Relational Algebra</td>
<td>✓</td>
</tr>
<tr>
<td>TDM</td>
<td>Segev</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>Temporal Relational Algebra</td>
<td>Lorentzos</td>
<td>Relational Algebra</td>
<td>✓</td>
</tr>
<tr>
<td>TempSQL</td>
<td>Yau</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>Time-By-Example</td>
<td>Tansel</td>
<td>QBE</td>
<td>✓</td>
</tr>
<tr>
<td>TOSQL</td>
<td>Ariav</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>TQuel</td>
<td>Snedgrass</td>
<td>Quel</td>
<td>✓</td>
</tr>
<tr>
<td>TSQL</td>
<td>Navathe</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>TSQL2</td>
<td>BCDM</td>
<td>SQL-92</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>ADM</td>
<td>Relational Algebra</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Bassounid</td>
<td>Quel</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Ben-Zvi</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>DM/T</td>
<td>Relational Algebra</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Gadia-2</td>
<td>Quel</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>RDM</td>
<td>IL</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>HRDM</td>
<td>Relational Algebra</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>McKenzie</td>
<td>Relational Algebra</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.5: Temporal relational query languages

<table>
<thead>
<tr>
<th>Name</th>
<th>Underlying data model</th>
<th>Based on</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATISSE</td>
<td>MATISSE</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>ODAPEX</td>
<td>ODAPEX</td>
<td>DAPEX</td>
<td>✓</td>
</tr>
<tr>
<td>SQL</td>
<td>IRIS</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>OQL</td>
<td>OVM</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>OQL/T</td>
<td>OSAM*/T</td>
<td>OSAM*/OQL</td>
<td>✓</td>
</tr>
<tr>
<td>Orion</td>
<td>Kim</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>PICQUERY+</td>
<td>TEDM</td>
<td>PICQUERY</td>
<td>✓</td>
</tr>
<tr>
<td>Postgresql</td>
<td>PostgreSQL</td>
<td>Quel</td>
<td>✓</td>
</tr>
<tr>
<td>TMQL</td>
<td>TMAD</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>TQL</td>
<td>TIGUKAT</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>TOOQL</td>
<td>TOODM</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>TOOSQL</td>
<td>TOODM</td>
<td>SQL</td>
<td>✓</td>
</tr>
<tr>
<td>VISION</td>
<td>Caruso</td>
<td>Metamorph</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sciore-1</td>
<td>Annotations</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sciore-2</td>
<td>EXTRA/EXCESS</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.6: Temporal object-oriented query languages

5.7. BIBLIOGRAPHIC NOTES


The book edited by Tansel et al. provides a snapshot of temporal database research as of 1993 [427]. Chapters of that book provide excellent surveys on temporal reasoning [291] and on temporal deductive databases [46]. The glossary that was initially prepared for that book has since been considerably expanded; the most recent version appeared in SIGMOD Record [230].

Other extant surveys include those on temporal data models [233, 406], temporal query languages [107, 281, 407], and temporal access methods [383]. A recent survey covers both temporal and real-time databases [328]; that survey contains references to all the data models discussed here (in Tables 5.1, 5.2, 5.5, and 5.6). The solution in Example 5.5 was discovered independently by Böhlen [64] and by Rosenthal, Abramovich, and Birger [90, 368]. The relative performance of many algorithms for coalescing has been compared [96].

Van Benthem’s book is an excellent introduction to temporal logic [438], as is an earlier one by Rescher [357]. An extensive literature has since developed.
5. OVERVIEW OF TEMPORAL DATABASES

While previous authors had mentioned various kinds of time, Snodgrass and Ahn showed that there were two principal types, valid time and transaction time [408]. The concepts of temporal specialization and generalization, which characterize correlations between the valid and transaction times of a fact, were introduced by Jensen and Snodgrass [231]. Later they summarized the general topic of the semantics of time-varying data, including logical and physical design [232].

Time has been added to many data models, in addition to the relational and object-oriented data models discussed in the chapter. The entity-relationship model, semantic data models, knowledge-based data models, and deductive database models have been extended. References to these data models may be found in the survey [328]. More information on the five models discussed in Section 5.4.3 is available: Segev [395], Sarda [385], HRDM [110], EDM [111], and Bhargava [56]. There are also several data models involving constraints [235, 255, 256]. The relational algebra may be extended to manipulate these constraints.

SQL-92 is described in Melton and Simon’s book [285]. References to the query languages mentioned in this chapter may be found in the survey [328].

We did not consider the related topic of temporal reasoning (also termed inferencing or rule-based search) [236, 263, 291, 414], which uses artificial intelligence techniques to perform more sophisticated analyses of temporal relationships, generally with much lower query processing efficiency. Also not included are knowledge representation languages, such as Telos [300] or TSOS [43], which, while supporting either valid or transaction time, or both, are not strictly query languages.

Oversimplifying, the history of temporal databases can be seen as comprising four overlapping phases. The first, from 1956 to 1985, focused on concept development, considering the multiple kinds of time and conceptual modeling. The second, from 1978 to 1994, contained two subphases: 1978 to 1990 saw the design of many relational temporal query languages, while 1990 to 1994 experienced the introduction of object-oriented temporal query languages. The third phase, from 1988 to the present, considered implementation aspects, in particular storage structures, operator algorithms, and temporal indexes. The final phase is one of consolidation, starting with the infrastructure workshop in 1993 and continuing to the present. In this phase a consensus glossary, query language test suite, and TSQL2 emerged. The number of temporal database papers continues to increase at a superlinear rate, as it has for the past two decades.

5.8 EXERCISES

5.8 Exercises

5.1. In Example 5.4, the repeat-until loop updates the Stop time.
   
   a. Revise the UPDATE statement to update the Start time instead. Argue the correctness of your solution.
   
   b. Combine the approaches to simultaneously update both the Start and Stop times.

5.2. Five ways were presented to perform coalescing: (1) updating the Stop time (Example 5.4), (2) SELECT DISTINCT (Example 5.5), (3) using a cursor (Example 5.6), (4) updating the Start time (Exercise 5.1a), and (5) updating both the Start and Stop times (Exercise 5.1b). What is the worst-case complexity of each of these five approaches, for n value-equivalent tuples? To determine their relative average-case efficiency, run each on a conventional DBMS.

5.3. The UA Office of Appointed Personnel is asked, “What is the maximum salary?”

   a. Give this query in SQL on a snapshot database, storing only the current information.
   
   b. Now that the salary history is stored, we’d like a history of the maximum salary over time. The problem, of course, is that SQL does not provide temporal aggregates. One way to do this is indirectly, by converting the snapshot aggregate query into a nonaggregate query, then converting that into a temporal query. The nonaggregate query finds those salaries for which a greater salary does not exist. Write this query in SQL, again, on a snapshot database.
   
   c. Now convert this latter query into a temporal query. This is quite challenging.
   
   d. How do you think this query could be expressed in TSQL2?
   
   e. Why doesn’t the trick in (c) work when asked to compute the average salary over time?

5.4. Show how, by using derived tables, i.e., FROM ( SELECT .... ) AS F, Example 5.5 can be expressed in a single SQL statement. Why doesn’t this transformation work for Example 5.4?
5.5. Remove the linked list from the code in Example 5.6, using ORDER BY Start. Run both versions on a conventional DBMS to determine their relative efficiency.

5.6. Pick your favorite application. Be creative.

   a. What aspects of this application require storing historical or future information?
   b. Specify an SQL schema for several historical relations, using columns of type DATE or TIMESTAMP.
   c. Populate these relations with sample data.
   d. Provide some interesting English prose queries, along with the results that should be returned when evaluated on the same data.
   e. Express your queries in SQL. Evaluate the difficulty of using SQL for your application.
   f. Discuss how valid and transaction time relate to your application. Which relations should support valid time, which should support transaction time, and which should support both?
   g. Extend your sample data to include both valid and transaction time.

Chapter 6

TSQIL2

The Temporal Structured Query Language, or TSQIL2, was designed by a committee of 18 researchers who had individually designed many of the languages listed in the previous chapter. The goal of TSQIL2 was to consolidate approaches to temporal data models and calculus-based query languages, to achieve a consensus extension to SQL-92 and an associated data model upon which future research could be based. Additionally, TSQIL2 is being incorporated into the evolving SQL3 standard.

6.1 Time Ontology

TSQIL2 uses a linear time structure, bounded on both ends. The origin is 18 billion years ago, when the Big Bang is thought to have occurred; the time line extends 18 billion years into the future.

The TSQIL2 time line is a discrete representation of the real time line, which can be considered to be discrete, dense, or continuous. The TSQIL2 time line consists of atomic (nondecomposable) chronons. Consecutive chronons may be grouped together into granules, with different groupings yielding distinct granularities. TSQIL2 allows a value of a temporal data type to be converted from one granularity to another.

TSQIL2 is carefully designed not to require choosing among the discrete, dense, and continuous time ontologies. Rather, TSQIL2 permits no question to be asked that would differentiate among these three ontologies. For example, it is not possible to ask if an instant a precedes an instant b. It is only possible to ask that question in terms of a specified granularity, such as seconds, days, or years. Different granularities could yield different answers to this question. Similarly, distance is in terms of a specified granularity, and is represented as an integral number of granules.
6.2 Data Model

TSQL2 employs a very simple underlying data model. This data model retains the simplicity and generality of the relational model. It has no illusions of being suitable for presentation, storage, or query evaluation. Instead, separate, representational data models, of equivalent expressive power, are employed for implementation and for ensuring high performance. Other presentational data models may be used to render time-varying behavior to the user or application. A coordinated suite of data models can achieve in concert goals that no single data model could attain.

This conceptual model, termed the Bitemporal Conceptual Data Model (BCDM), timestamps tuples with bitemporal elements, which are sets of bitemporal chronons. Each bitemporal chronon represents a tiny rectangle in valid-time/transaction-time space. Because no value-equivalent tuples are allowed in a bitemporal relation instance, the full time history of a fact is contained in a single tuple. Equivalently, we say that the BCDM is a coalesced data model. In Table 5.3, the conceptual temporal data model occupies the entry corresponding to timestamping tuples with valid-time elements, and it occupies the entry in Table 5.4 corresponding to timestamping tuples with transaction-time elements.

Example 6.1 A bitemporal relation

Consider an Employee relation recording information such as

"Jake works for the shipping department." We assume that the granularity of chronons is 1 day for both valid time and transaction time, and the period of interest is some given month in a given year (e.g., the integer 15 in a timestamp represents the date June 15, 1996).

Figure 6.1 shows how the bitemporal element in an employee's department tuple changes. In graphical representations of bitemporal space, we choose the x-axis as the transaction-time dimension, and the y-axis as the valid-time dimension. Hence, the ordered pair (t, v) represents the bitemporal chronon with transaction time t and valid time v.

Employee Jake was hired by the company as temporary help in the shipping department for the period from time 10 to time 15, and this fact became current in the database at time 5. This is shown in Figure 6.1(a). The arrows pointing to the right signify that the tuple has not been logically deleted; it continues through to the transaction time until changed (U.C.)

Figure 6.1(b) shows a correction. The personnel department discovers that Jake had really been hired from time 5 to time 20, and the database is corrected beginning at time 10. Later, the personnel department is informed that the correction was itself incorrect; Jake really was hired for the original time period, time 10 to time 15, and the correction took effect in the database at time 15. This is shown in Figure 6.1(c). Lastly, Figure 6.1(d) shows the result of three updates to the relation, all of which become current starting at time 20. (The same transaction could have caused all three of these updates.) While the period of validity was correct, it was discovered that Jake was not in the shipping department, but in the loading department. Consequently, the fact (Jake,Ship) is removed from the current state and the fact (Jake,Load) is inserted. A new employee, Kate, is hired for the shipping department for the period from time 25 to time 28.

We note that the number of bitemporal chronons in a given bitemporal element is the area enclosed by the bitemporal element. The bitemporal element for (Jake,Ship) contains 140 bitemporal chronons.

The actual bitemporal relation corresponding to the graphical representation in Figure 6.1(d) is shown in Figure 6.2. This relation contains three facts. The timestamp attribute T shows each transaction-time chronon associated with each valid-time chronon as a set of ordered pairs.

It is possible to demonstrate equivalence mappings between the conceptual model and several representational models. Mappings have been demonstrated for five bitemporal data models: BeimZvi (five timestamps per tuple), Bhargava (attribute timestamping, illustrated in the previous chapter), Jensen (tuple timestamping with a single transaction chronon), McKenzie (transaction timestamping of states, valid timestamping of attributes), and Snodgrass (tuple timestamping, with period valid and transaction times). This equivalence is based on snapshot equivalence, which says
time-varying semantics can be considered in isolation, utilizing different data models. Semantics, specifically as determined by logical database design, is expressed in the conceptual model. Multiple presentation formats are available, as different applications require different ways of viewing the data. The storage and processing of bitemporal relations are performed in a data model that emphasizes efficiency, as discussed in the next chapter.

6.3 Language Constructs

We now turn to the statements available in TSQL2.

6.3.1 Schema Definition

This language is a strict superset of SQL-92, and so it supports conventional relations in all their grandeur. To explore the temporal features of TSQL2, we'll need a temporal relation. Envision a patient database at a doctor's office. Included in this database is information on the drugs prescribed to each patient.

Example 6.2 Define the Prescription relation

```
CREATE TABLE Prescription (Name CHAR(30),
Physician CHAR(30), Drug CHAR(30), Dosage CHAR(30),
Frequency INTERVAL MINUTE)

AS VALID STATE DAY AND TRANSACTION
```

The Name column specifies the patient's name. The Frequency is the number of minutes between drug administrations.

The AS clause is new in TSQL2. The valid time specifies the period(s) during which the drug was prescribed. The transaction time specifies when this information was recorded as current in the database. Tuples that have not been updated or deleted will have a transaction time that includes now.

The valid time has a granularity of 1 day. The granularity of the transaction time is system-dependent, but most likely will be no coarser than a millisecond, to differentiate consecutive transactions.

The Prescription relation is a bitemporal state relation, as it includes both kinds of time. There are six kinds of relations:

- snapshot relations, which have no temporal support
- valid-time state relations, specified with AS VALID STATE (STATE is optional)
valid-time event relations, specified with **AS VALID EVENT**

- transaction-time relations, specified with **AS TRANSACTION**

- bitemporal state relations, specified with **AS VALID STATE AND TRANSACTION**

- bitemporal event relations, specified with **AS VALID EVENT AND TRANSACTION**

The type of a relation can be changed at any time, using the **ALTER TABLE** statement.

### 6.3.2 The SELECT Statement

To obtain a snapshot result from a temporal relation, specify the new reserved word **SNAPSHOT**.

**Example 6.3** Who has been prescribed drugs?

```sql
SELECT SNAPSHOT Name
FROM Prescription
```

This will return the names of all patients who have been prescribed drugs, now or in the past.

**Example 6.4** Who is or was taking the drug Proventil?

```sql
SELECT SNAPSHOT Name
FROM Prescription
WHERE Drug = 'Proventil'
```

Again, a simple list of names results.

The history can also be requested:

**Example 6.5** Who has been prescribed drugs, and when?

```sql
SELECT Name
FROM Prescription
```

The default is to return the history, so omitting **SNAPSHOT** does the trick. TSQL2 performs automatic coalescing, so the result is a set of tuples, each associated with one or more maximal periods, during which time the patient was prescribed at least one drug.

When more than one correlation name is mentioned, the default is to identify those moments when all of the underlying tuples are valid. We are interested in the interactions of Proventil with other drugs.

### Example 6.6 What drugs have been prescribed with Proventil?

```sql
SELECT P1.Name, P2.Drug
FROM Prescription AS P1, Prescription AS P2
WHERE P1.Drug = 'Proventil' AND P2.Drug <> 'Proventil'
AND P1.Name = P2.Name
```

The result is a set of tuples, each specifying a patient and a drug, along with the maximal period(e) during which both that drug and Proventil were prescribed to that patient.

### 6.3.3 Restructuring

One of the most powerful constructs of TSQL2 is **restructuring**. Whereas TSQL2 automatically performs coalescing on the result of a query, restructuring in the **FROM** clause allows coalescing to be performed on the underlying tuples.

**Example 6.7** Who has been on a drug for more than a total of six months?

```sql
SELECT Name, Drug
FROM Prescription(Name, Drug) AS P
WHERE CAST(VALID(P) AS INTERVAL MONTH) > INTERVAL '6' MONTH
```

Notice that the **FROM** clause mentions in parentheses several of the attributes of the Prescription relation. This clause projects out the Name and Drug attributes, then coalesces the result, which is then manipulated in the remainder of the query. By restructuring on Name and Drug, the timestamp associated with each name-drug pair indicates the maximal period(s) when that patient was prescribed that drug, independent of the prescribing physician, the dosage, or the frequency. Hence, a single pair may be computed from many pairs of the underlying Prescription relation. The other attributes are not available via **P**.

The new **VALID(P)** construct returns the valid-time element (set of maximal periods) associated with P. Then, the **CAST** operator converts it to the type **INTERVAL MONTH** by summing the durations (in months) of each of the maximal periods. This computes the total number of months that patient has been prescribed that drug, ignoring gaps when the drug was not prescribed. This total is compared with the interval constant 6 months.

The result is a relation with two columns, the patient's name and the drug, along with a timestamp specifying when that drug was prescribed.
6. TSQL2

However, only drugs that have been prescribed for at least a total of 6 months will appear in the result.

Correlation names can be coupled (one correlation name is defined in terms of another correlation name) with further restructuring.

Example 6.8 Who has been on Proventil throughout their drug regime?

```
SELECT SNAPSHOT P1.Name
FROM Prescription(Name) AS P1, P1(Drug) AS P2
WHERE P2.Drug = 'Proventil' AND VALID(P2) = VALID(P1)
```

The first portion of the FROM clause projects out just the Name and couleses, with the resulting timestamp indicating when that patient was prescribed any drug, by any physician, at any frequency and dosage. The correlation name P2 is defined in terms of P1. P2 adds the Drug attribute, and hence is similar to the P correlation name defined in the previous query, as it is restructured on both Name (from P1) and Drug. Also, since P2 was defined in terms of P1, TSQL2 ensures that the common attributes—in this case, Name—have the same value for both. So P1 captures when the patient was prescribed any drug, and P2 captures when that patient was prescribed the drug Proventil. The WHERE clause stipulates that the valid times (both of which are sets of maximal periods) must be the same, meaning that whenever the patient was prescribed any drug, she was prescribed Proventil.

It is always the case that VALID(P1) CONTAINS VALID(P2), by virtue of P2 being a further restructuring of P1, but only for some patient names. The two temporal elements be equal.

Finally, note that SNAPSHOT is specified, so the result is simply a set of patient names.

Interestingly, both restructuring and coupled correlation names are syntactic sugar. The above query can be rephrased without using either construct.

Example 6.9 The same query as Example 6.8, but without using restructuring or coupled correlation names

```
SELECT SNAPSHOT P1.Name
FROM (SELECT Name FROM Prescription) AS P1,
     (SELECT Name, Drug FROM Prescription) AS P2
WHERE P1.Name = P2.Name AND P2.Drug = 'Proventil'
     AND VALID(P2) = VALID(P1)
```

Hence, restructuring is effectively a nested SELECT clause, to perform the projection and coalescing. Coupling correlation names implies equality of their shared attributes.

6.3. LANGUAGE CONSTRUCTS

Intuitively, P1 ranges over Name and is timestamped with a temporal element indicating when the value of the Name attribute remained constant. P2 ranges over different values of Drug, with the timestamp of P2 being a subset (proper or otherwise) of the timestamp of P1.

6.3.4 Partitioning

Regardless of whether the correlation name has been restructured, the timestamp is still a temporal element. Often we wish to examine the constituent maximal periods of the timestamp. This can be accomplished by partitioning, which is denoted with (PERIOD).

Example 6.10 Who has been on the same drug for more than 6 consecutive months?

```
SELECT SNAPSHOT Name, Drug, VALID(P)
FROM Prescription(Name, Drug)(PERIOD) AS P
WHERE CAST(VALID(P) AS INTERVAL MONTH) > INTERVAL '6' MONTH
```

It is useful to compare the English and TSQL2 versions of this query with those of the previous query. We first restructure on Name and Drug, computing a temporal element for each name-drug pair. Then each maximal period is extracted, so that P contains potentially many tuples with identical name-drug values, but timestamped with different maximal periods. VALID(P) is then of type period, rather than a temporal element. The WHERE clause considers the duration of an individual period, rather than the total of all periods, thus capturing the notion of 6 consecutive months.

The result is a snapshot relation, with a PERIOD value in the last attribute. This result may contain several tuples with the same Name and Drug values, but with different values for the third attribute.

Alternatively, we could have requested a valid-time relation:

Example 6.11 Who has been on the same drug for more than 6 consecutive months?

```
SELECT Name, Drug
FROM Prescription(Name, Drug)(PERIOD) AS P
WHERE CAST(VALID(P) AS INTERVAL MONTH) > INTERVAL '6' MONTH
```

In this case, only one tuple for each name-drug pair would be returned, with an associated temporal element timestamp, containing only those maximal periods of duration greater than 6 months.
Partitioning, however, is not syntactic sugar. The intermediate relation (the result of the FROM clause) violates the data model, since it produces value-equivalent tuples. Note that the underlying and resulting relations are always coalesced, so this violation is isolated to the query evaluation.

To summarize, \texttt{VALID}(P) evaluates to the timestamp associated with the correlation name \texttt{P}. For state relations associated with unpartitioned correlation names (whether or not restructured), this evaluates to a temporal element. For state relations partitioned by \texttt{(PERIOD)}, this evaluates to a single period.

In the above example, \texttt{VALID}(P) was used in the \texttt{SELECT} list. This is permitted because \texttt{P} was partitioned. An attribute cannot be of type temporal element, and so this usage would not have been correct if \texttt{P} was not partitioned.

### 6.3.5 The \texttt{VALID} Clause

To this point, the timestamp of the resulting tuples has defaulted to the intersection of the timestamps of the underlying tuples associated with the correlation name(s). This default can be overridden via a \texttt{VALID} clause.

**Example 6.12** What drugs was Melanie prescribed during 1996?

\[
\begin{align*}
\text{SELECT} & \quad \text{Drug} \\
& \quad \text{VALID INTERSECT(VALID(Prescription), PERIOD ['1996'] DAY)} \\
\text{FROM} & \quad \text{Prescription} \\
\text{WHERE} & \quad \text{Name = 'Melanie'}
\end{align*}
\]

The result is a list of drugs, each associated with a set of the periods during 1996 during which the drug was prescribed to Melanie. Those drugs that were prescribed only before or after 1996 will not be included, because the intersection will result in an empty temporal element, which is disallowed as a timestamp. This intersection is between a temporal element and a specified period. It is possible that multiple periods of the temporal element will intersect the year 1996, with the result containing multiple periods.

### 6.3.6 The Modification Statements

The SQL modification statements, \texttt{INSERT}, \texttt{DELETE}, and \texttt{UPDATE}, apply to temporal relations.

### 6.3. LANGUAGE CONSTRUCTS

**Example 6.13** Insert a prescription today

\[
\begin{align*}
\text{INSERT INTO} & \quad \text{Prescription} \\
\text{VALUES} & \quad ('Melanie', 'Dr. Beren', 'Proventil', '100mg', \\
& \quad \text{INTERVAL '8:00' MINUTE})
\end{align*}
\]

In this example, we didn't specify a timestamp, so the timestamp defaults to \texttt{VALID PERIOD(CURRENT_TIMESTAMP, NOBIND(CURRENT_TIMESTAMP))}.

Assume that the clock on the wall says that it is currently July 9, 1996, at 1:30 PM. The prescription starts on \texttt{DATE '1996-07-09'} (it is a date because that is the valid-time granularity of the Prescription relation). It is valid until \texttt{NOBIND(CURRENT_TIMESTAMP)}, which is the way in TSQL2 that you specify storing the variable now. This value is bound whenever it is retrieved from the database. So if we evaluated the following query,

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{Prescription}
\end{align*}
\]

we would see that that prescription went from July 9 to July 9. If we executed the same query 3 days from now (i.e., on July 12), we would see that the prescription went from July 9 to July 12. This is because we don't know what will happen in the future, so the default is to indicate that the tuple was valid until now.

Of course, most prescriptions are not open-ended, and so a terminating date is generally known.

**Example 6.14** Insert a prescription with a known period of validity.

\[
\begin{align*}
\text{INSERT INTO} & \quad \text{Prescription} \\
\text{VALUES} & \quad ('Melanie', 'Dr. Beren', 'Proventil', '100mg', \\
& \quad \text{INTERVAL '8:00' MINUTE}) \\
\text{VALID PERIOD} & \quad ['1996-01-01 - 1996-06-30']
\end{align*}
\]

We use the \texttt{VALID} clause, introduced earlier for the \texttt{SELECT} statement.

Since TSQL2 automatically coalesces, if there was a value-equivalent tuple already in the relation, its temporal element timestamp would be coalesced with the inserted period. Only if a value-equivalent tuple was not already present would this \texttt{INSERT} statement result in a new tuple being added to the relation.

In both cases, the transaction time of the new tuple has a value identical to the valid-time default. That is, the tuple was inserted at \texttt{CURRENT_TIMESTAMP} and is in the database through \texttt{U.C.}; we have no way
of knowing whether it will still be in the database tomorrow. It is not possible to specify a transaction time, as that semantics of transaction time must be ensured by the DBMS.

The VALID clause can also be used in the DELETE and UPDATE statements:

Example 6.15  Melanie wasn’t prescribed anything for June 1996

\[
\text{DELETE FROM Prescription}
\]
\[
\text{WHERE Name = ‘Melanie’}
\]
\[
\text{VALID PERIOD } [1996-06-01 - 1996-06-30]
\]

The month of June 1996 is removed from the timestamp of each tuple for the patient named Melanie. Some tuples might not be affected at all (if they do not overlap June 1996), some might be removed entirely (if they were valid only for June 1996, or a part thereof), and some might have a portion of their timestamp removed (if a portion of the timestamp overlapped June 1996). TSQL2 takes care of the details in these three cases.

The semantics of the UPDATE statement is quite similar, but the mechanics are more involved:

Example 6.16  Change the Proventil dosage to 50 mg

\[
\text{UPDATE Prescription}
\]
\[
\text{SET Dosage TO ‘50 mg’}
\]
\[
\text{WHERE Name = ‘Melanie’ AND Drug = ‘Proventil’}
\]

This changes all current and future Proventil prescriptions to a dosage of 50 milligrams. Prescriptions valid in the past are unaffected. If there were no tuples currently existing with such a dosage, this might actually cause a tuple to be inserted into the relation.

Example 6.17  Change the dosage for March through May

\[
\text{UPDATE Prescription}
\]
\[
\text{SET Dosage TO ‘50 mg’}
\]
\[
\text{VALID PERIOD } [1996-03-01 - 1996-05-30]
\]
\[
\text{WHERE Name = ‘Melanie’ AND Drug = ‘Proventil’}
\]

Here the dosages before March 1996 and after May 1996 are unaffected.

6.3.7  Event Relations

To this point, we have considered only the Prescription relation, which is a bitemporal state relation, recording facts that are true over time. Event relations record instantaneous events. Event relations are timestamped with

\section{6.3. LANGUAGE CONSTRUCTS}

instant sets, which are simply sets of instants. Each tuple identifies a particular kind of event, and the timestamp of that tuple specifies the instant(s) when that event occurred.

Example 6.18  Define the LabTest event relation

\[
\text{CREATE TABLE LabTest (Name CHAR(30), Physician CHAR(30), TestID INTEGER)}
\]
\[
\text{AS VALID EVENT HOUR AND TRANSACTION}
\]

A lab test occurs at a particular hour (we are not interested in a finer granularity) and is ordered by a physician for a patient. TestID identifies a particular kind of test (e.g., blood test), which might possibly be administered several times to a particular patient.

Event relations can also be restructured and partitioned.

Example 6.19  Were any patients the sole receivers of tests ordered by a physician?

\[
\text{SELECT L1.Name, L2.Physician}
\]
\[
\text{FROM LabTest(Name AS L1, L1(Physician) AS L2,}
\]
\[
\text{LabTest(Physician) AS L3}
\]
\[
\text{WHERE VALID(L1) = VALID(L2) AND L2.Physician = L3.Physician}
\]
\[
\text{AND VALID(L1) = VALID(L3)}
\]

\text{VALID(L1)} is an event set containing all tests done on a particular patient, because of the restructuring on Name. Thus, the event sets of all tuples with the same Name are coalesced into a single event set. \text{VALID(L2)} is an event set containing all tests done on the same patient as in \text{L1}, but ordered by a particular physician. In both cases, the TestID is ignored. Because of the semantics of coupled correlation names, it must be the case that \text{VALID(L1) CONTAINS VALID(L2)}. Finally, \text{VALID(L3)} is an event set containing all tests ordered by a particular physician, ignoring the patient.

The predicate \text{VALID(L1) = VALID(L2)} requires that all tests administered to the patient be ordered by \text{L2.Physician}. The predicate \text{VALID(L1) = VALID(L3)} requires that all those tests be ordered by \text{L3.Physician}.

6.3.8  Transaction-Time Support

To this point, we haven’t discussed the implications of the Prescription relation supporting transaction time. In particular, all the queries have implicitly applied to the current state of the relation, ignoring older, corrected tuples.
6.3. LANGUAGE CONSTRUCTS

Example 6.20  What is Melanie's prescription history?

SELECT Drug
FROM Prescription
WHERE Name = 'Melanie'

This returns the prescription history as best known, taking into account all corrections that have been entered.

We can roll back the database to its contents stored at a previous point in time.

Example 6.21  What did the physician believe on June 1, 1996, was Melanie's prescription history?

SELECT Drug
FROM Prescription AS P
WHERE Name = 'Melanie'
AND TRANSACTION(P) OVERLAPS '1996-06-01'

TRANSACTION(P) is allowed only on transaction-time and bitemporal relations, and returns the maximal period in transaction time when the values of the attributes and the valid time associated with the tuple (if present) remained constant.

The default predicate is

TRANSACTION(P) OVERLAPS CURRENT_TIMESTAMP

In the above example, we specified a different transaction time. Note that the result could contain drugs that we determined later had not in fact been prescribed to Melanie.

Auditing can be done on previously corrected data.

Example 6.22  When was Melanie's data, valid on June 1, 1996, last corrected?

SELECT SNAPSHOT BEGIN(TRANSACTION(P2))
FROM Prescription AS P P2
WHERE P1.Name = 'Melanie' AND p2.Name = 'Melanie'
AND VALID(P1) OVERLAPS '1996-06-01'
AND VALID(P2) OVERLAPS '1996-06-01'
AND TRANSACTION(P1) MEETS TRANSACTION(P2)

We are interested in data concerning the state of the modeled reality for Melanie on June 1, 1996, and so both P1 and P2 are required to be valid on that date, with a Name of Melanie. The predicate TRANSACTION(P1)
MEETS TRANSACTION(P2) says that the tuple associated with P1 was corrected, with the new value of one of the other attributes recorded in the tuple associated with P2. The actual transaction performing the update of this tuple has a transaction time of BEGIN(TRANSACTION(P2)). Note that the tuple associated with P2 may also be incorrect, in which case there will be yet another tuple whose transaction time period meets that of P2.

Note that the transaction time of tuples is supplied by the system. Hence, there is no transaction clause that mirrors the valid clause discussed above for SELECT and modification statements.

6.3.9  Aggregates

SQL-92 supports the aggregates MIN, MAX, COUNT, SUM, and AVG. These aggregates return time-varying results when applied to temporal relations.

Example 6.23  How many drugs has Melanie been taking?

SELECT COUNT(*)
FROM Prescription
WHERE Name = 'Melanie'

This is a conventional SQL-92 query applied to a bitemporal state relation. The current valid-time state is queried; tuples whose transaction time does not overlap now are ignored. A valid-time state relation will be returned, providing a time-varying count of the number of prescriptions valid at any point in time.

Aggregates may also have a GROUP BY clause:

Example 6.24  How many people were taking each drug?

SELECT Drug, COUNT(*)
FROM Prescription
GROUP BY Drug

Again, the result is a valid-time state relation, showing the history of the count for each drug.

TSQL2 adds one aggregate, RISING, which evaluates to the longest period during which the specified attribute was monotonically rising.

Example 6.25  When was Melanie's Proventil dosage rising the longest?

SELECT SNAPSHOT RISING(Dosage)
FROM Prescription
WHERE Name = 'Melanie' AND Drug = 'Proventil'
This query returns a set of periods, indicating those stretches of time when the dosage was rising. If the dosage rose (or stayed level), then fell, then rose, the result would consist of two periods.

### 6.3.10 Schema Evolution and Versioning

SQL permits the schema to be changed by using the `ALTER` statement, termed schema evolution. Only one schema is in effect at any time; a schema change causes the previous schema to be lost. In TSQL2, if the relation has transaction-time support (i.e., if it is a transaction-time or bitemporal relation), then the schema is versioned for that relation. Effectively, the schema itself becomes a set of transaction-time relations.

The `Prescription` relation now has five attributes. A sixth attribute is later added:

**Example 6.26** Add a column to the `Prescription` relation on August 20, 1996

```sql
ALTER TABLE Prescription ADD COLUMN Identifier INTEGER
```

Both the previous schema, with five attributes, and the new schema, with six attributes, are retained. Since data in a relation that supports transaction-time cannot be modified after it is stored (since such relations are append-only), a schema change can affect only data written by the transaction effecting the schema change, or by future transactions.

Legacy applications written before August 20 may want to use the old schema in force when the application was developed. They can do so by specifying SET SCHEMA DATE '1996-08-19'. This allows data written after (or before) that date to be viewed as of the schema in effect on that date. TSQL2 transforms data of different schemas into the schema associated with the query. In the case of the one attribute being added, using the old schema simply means that that attribute is unavailable to the query. Some schema changes cannot be accommodated; an example is a change that splits one relation into two. In such situations, the query's schema must match the data's schema.

### 6.4 Other Constructs

We end with a brief discussion of some additional features of TSQL2.

A surrogate is a unique value that can be compared for equality but is otherwise not visible to users. Surrogates are useful for identifying objects when the primary key of the object is time-varying. TSQL2 adds a new

**SURROGATE**, as well as a unary function, **NEW**, that supplies a surrogate value that has never been used before.

A temporal granularity is a partitioning of the time line into granules; examples include seconds, hours, academic semesters, and fiscal quarters. An instant is represented by an integral number of granules from an identified granularity anchor, which is a fixed point on the time line. Granularities are provided by calendars, which also supply mappings between granularities. The granularities together form a lattice, which ensures that it is possible to convert a temporal value in any granularity into any other granularity.

Calendars are gathered into calendric systems, which can be then selected by the user. The default is the provided SQL-92 calendric system, which defines the granularities already present in SQL-92: **SECOND, MINUTE, HOUR, DAY, MONTH, and YEAR**.

Operands of predicates (e.g., **PRECEDES**) must be comparable, that is, must have the same granularity. The predicate is performed at this implicit granularity. The user can change the granularity of the operands, and thus of the operation.

Temporal indeterminacy is "don't know when" information. Examples include "between 2 PM and 4 PM" and "around the middle of August." Note that in both cases the event is known to have happened; it is not certain when, or more precisely, exactly when, the event occurred. Temporal indeterminacy interacts closely with granularity. Some time during the first week in January 1996 is indeterminate at a granularity of days, but it is determinate at the granularity of weeks. Section 14.3 discusses temporal indeterminacy in the general context of uncertainty in databases.

There are two conversion functions available: **CAST** and **SCALE**. CAST always results in a determinate value; SCALE preserves as much information as possible, which may require indeterminacy. Scaling an instant to a coarser granularity may cause an indeterminate instant to become determinate; scaling to a finer granularity always introduces indeterminacy. As an example, **CAST(TIMESTAMP '04-19-1996 15:24:00 AS DAY) results in '04-19-1996'; the same value results if SCALE is used. However, CAST(DAY '04-19-1996 AS SECOND) results in the first second of that day, SCALE will result in an indeterminate instant occurring during an unspecified second of the 86,400 seconds of that day.**

SQL-92 includes the nullary functions CURRENT.DATE, CURRENT.TIME, and CURRENT.TIMESTAMP, but doesn't allow these variables to be stored in relations. As discussed above, TSQL2 does allow these values to be stored, via NOBIND. In addition, TSQL2 supports more general versions
of these now variables—specifically, now-relative values such as 'now - 3' days as well as indeterminate versions.

Relations supporting transaction time are append-only, growing monotonically. This behavior can cause problems. The most obvious ramification is that data could outgrow the available storage media. Even if the data fits, more data means slower querying and update. Finally, many countries have strict laws that necessitate the physical deletion of old data, to prevent particular data from being retained longer than a specified time interval. Hence, there is the need to remove data that is no longer useful from the database.

Vacuuming is the process of removing obsolete data. Note that data with a transaction end time of now can easily be removed by simply using the DELETE statement. If the relation has transaction-time support, however, this data will be retained, with a transaction end time of the transaction in which the delete occurred. A separate vacuuming statement is available in TSQL2 to eliminate data with a transaction end time before a specified date.

6.5 Summary

Here are the major concepts underlying TSQL2:

- **State relations** are timestamped with temporal elements, which are sets of periods.

- Conventional (non-time-varying) relations can be derived from time-varying relations by specifying SNAPSHOT in the SELECT clause. Conventional relations can participate along with time-varying relations in a query or modification statement.

- **Periods** are anchored durations of time. This is a new data type, augmenting SQL's datetimes and intervals.

- **Restructuring** merges the timestamps of value-equivalent tuples, and is specified by listing column names in the FROM clause.

- **Coupled correlation names** permit a further restructuring on additional columns, while ensuring that the tuples associated with the two correlation names agree on the values of the original coalescing columns.

- **Partitioning** extracts maximal period(s) from a valid-time element timestamp for a tuple, and is specified by (PERIOD) in the FROM clause.

6.6 Bibliographic Notes

- **Valid-time selection** enables tuples to be selected by predicates on their timestamps, within the WHERE clause.

- **Valid-time projection** specifies the period of validity of a derived relation, via the VALID clause.

- **Event relations** are timestamped with sets of instants.

- **Bitemporal relations** are timestamped with both valid time and transaction time.

- **Transaction-time selection** permits specification of previous versions.

- **Time-varying aggregates** can be computed. Grouping can be over columns or over tuple timestamps.

- **Schema versioning** allows relations timestamped with transaction time to be accessed and modified through previous schemas, thereby supporting legacy applications.

- **Surrogates** identify objects when the primary key is time-varying.

- **A granularity** is a partitioning of the time line.

- **Calendars** provide a collection of granularities and conversions between those granularities.

- **Temporal indeterminacy** allows "don't know precisely when" information to be recorded and queried.

- **Now-relative** times are bound during query evaluation.

- Relations timestamped with transaction time may be vacuumed to remove old versions.

6.6 Bibliographic Notes

TSQL2 is described thoroughly in a book devoted to the language [409]. The TSQL2 data model is further elaborated in [234]. Constructs in the language are being incorporated into a new part of SQL3 called SQL/Temporal [284]. As of January 1997, SQL/Temporal includes the PERIOD data type.
6.7 Exercises

6.1. This exercise concerns the personnel database introduced in Chapter 5 (see Exercise 5.3).

   a. Define the Employee relation as a bitemporal table using TSQL2's `CREATE TABLE` statement.
   b. Express the following in TSQL2 on this relation:
      i. What is the history of the maximum salary?
      ii. What is the history of the average salary?

6.2. Show, with a concrete example relation, how a DELETE statement with a specified valid time can

   a. not affect some tuples at all
   b. remove some tuples entirely
   c. remove a portion of some tuples

Show this relation before and after the update (see Example 6.15).

6.3. Show how the UPDATE statement in Example 6.17, executed on September 15, 1996, affects the following Prescription relation:

<table>
<thead>
<tr>
<th>Name</th>
<th>Drug</th>
<th>Valid Time</th>
<th>Transaction Time</th>
</tr>
</thead>
</table>

6.4. Provide an example where

   `CAST(VALID(A) AS ?) PRECEDES CAST(VALID(B) AS ?)`

   could yield different results for different granularities replacing the "?".

6.5. From the Employee relation shown in Figure 6.2, give the following queries in TSQL2 and provide the resulting relations:

   a. As known on June 7, in which department and for what time was Jake working?
   b. As known on June 12, ... 
   c. As known on June 17, ... 
   d. As best known, ...

6.6. This exercise concerns the personnel database introduced in Chapter 5 (see Exercise 5.6).

   a. Translate your relational schema to TSQL2.
   b. Express your original queries in TSQL2. Evaluate the use of TSQL2 for your application.
   c. Show how restructuring, partitioning, and aggregation could be used in your application.
   d. Add sample data and queries, expressed in English and in TSQL2, with results shown, for the following aspects:
      i. temporal indeterminacy
      ii. temporal granularity
      iii. event relations

6.7. Section 5.1 illustrated how a temporal query language can greatly simplify temporal queries. However, when a relation is ALTERed to be a temporal table, the existing application breaks: queries that before returned a snapshot relation of the current state now return a time-varying valid-time relation. An alternative to using ALTER is to retain the original Employee relation, recoding the current situation, and create a new table, EmployeeHistory, which stores the historical information.

   a. Create this new table using TSQL2.
   b. We wish to avoid changing the original personnel application to maintain this new table. Define rules, using Starburst, Oracle, DB2, or Chimera syntax, that can automatically maintain the EmployeeHistory relation in the presence of modifications to the snapshot Employee relation.

6.8. TSQL2, being an extension of SQL-92, did not consider triggers. Revisit the description of DB2 triggers in Section 2.3, and for each element of the syntax and the semantics, discuss what changes are necessary, if any, when the underlying table is a temporal table. Consider valid-time and transaction-time support separately.